SEATTLE COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Reference Manual

Microsoft® MS™-DOS

Operating System

Programmer's Reference Manual

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® MS™-DOS

Operating System

Programmer's Reference Manual

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the Programmer's Reference Manual on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

(C) Microsoft Corporation 1981, 1983

If you have comments about this product or this manual, please complete the Comment Form at the back of the manual and return it to Seattle Computer Products, 1114 Industry Drive, Seattle WA 98188.

Microsoft is a registered trademark of Microsoft Corporation.

MS is a registered trademark of Microsoft Corporation.

CP/M is a registered trademark of Digital Research, Inc.

Document No. 900-001799

System Requirements

Disk drive(s)

One disk drive if and only if output is sent to the same physical disk from which the input was taken. None of the programs allows time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

General Introduction

Chapter 1 System Calls

1.1 Introduction 1-1
1.2 Programming Considerations 1-1
1.2.1 Calling From Macro Assembler 1-1
1.2.2 Calling From a High-Level Language 1-1
1.2.3 Returning Control to MS-DOS 1-2
1.2.4 Console and Printer Input/Output Calls 1-3
1.2.5 Disk I/O System Calls 1-3
1.3 File Control Block (FCB) 1-3
1.3.1 Fields of the FCB 1-4
1.3.2 Extended FCB 1-6
1.3.3 Directory Entry 1-6
1.3.4 Fields of the FCB 1-7
1.4 System Call Descriptions 1-9
1.4.1 Programming Examples 1-10
1.5 Xenix-Compatible Calls 1-11
1.6 Interrupts 1-14
20H Program Terminate 1-16
21H Function Request 1-18
22H Terminate Address 1-19
23H CONTROL-C Exit Address 1-19
24H Fatal Error Abort Address 1-20
25H Absolute Disk Read 1-23
26H Absolute Disk Write 1-25
27H Terminate But Stay Resident 1-27
1.7 Function Requests 1-28
1.7.1 CP/M-Compatible Calling Sequence 1-28
1.7.2 Treatment of Registers 1-29

Function Requests

00H Terminate Program 1-33
01H Read Keyboard and Echo 1-34
02H Display Character 1-35
03H Auxiliary Input 1-36
04H Auxiliary Output 1-37
05H Print Character 1-38
06H Direct Console I/O 1-40
07H Direct Console Input 1-42
08H Read Keyboard 1-43
09H Display String 1-44
0AH Buffered Keyboard Input 1-45
0BH Check Keyboard Status 1-47
0CH Flush Buffer, Read Keyboard 1-48
0DH Disk Reset 1-49
0EH Select Disk 1-50
0FH Open File 1-50
10H Close File 1-53
11H Search for First Entry 1-55
12H Search for Next Entry 1-57
13H Delete File 1-59

14H Sequential Read 1-61
15H Sequential Write 1-63
16H Create File 1-65
17H Rename File 1-67
19H Current Disk 1-69
1AH Set Disk Transfer Address . . 1-70
21H Random Read 1-72
22H Random Write 1-74
23H File Size 1-76
24H Set Relative Record 1-78
25H Set Vector 1-80
27H Random Block Read 1-81
28H Random Block Write 1-84
29H Parse File Name 1-87
2AH Get Date 1-90
2BH Set Date 1-92
2CH Get Time 1-94
2DH Set Time 1-95
2EH Set/Reset Verify Flag 1-97
2FH Get Disk Transfer Address . . 1-99
30H Get DOS Version Number 1-100
31H Keep Process 1-101
33H CONTROL-C Check 1-102
35H Get Interrupt Vector 1-104
36H Get Disk Free Space 1-105
38H Return Country-Dependent Information 1-106
39H Create Sub-Directory 1-109
3AH Remove a Directory Entry . . . 1-110
3BH Change Current Directory . . . 1-111
3CH Create a File 1-112
3DH Open a File 1-112
3EH Close a File Handle 1-115
3FH Read From File/Device 1-116
40H Write to a File/Device 1-117
41H Delete a Directory Entry . . . 1-118
42H Move a File Pointer 1-119
43H Change Attributes 1-120
44H I/O Control for Devices . . . 1-121
45H Duplicate a File Handle . . . 1-125
46H Force a Duplicate of a Handle 1-126
47H Return Text of Current Directory 1-127
48H Allocate Memory 1-128
49H Free Allocated Memory 1-129
4AH Modify Allocated Memory Blocks 1-130
4BH Load and Execute a Program . . 1-131
4CH Terminate a Process 1-134
4DH Retrieve the Return Code of a Child 1-135
4EH Find Match File 1-136
4FH Step Through a Directory Matching Files 1-138
54H Return Current Setting of Verify 1-139
56H Move a Directory Entry 1-140
57H Get/Set Date/Time of File . . 1-141

1.8 Macro Definitions for MS-DOS System

Call Examples (00H-57H) 1-142
1.9 Extended Example of MS-DOS System Calls 1-149

Chapter 2 MS-DOS 2.0 Device Drivers

2.1 What is a Device Driver? 2-1
2.2 Device Headers 2-3
2.2.1 Pointer to Next Device Field . . . 2-3
2.2.2 Attribute Field 2-4
2.2.3 Strategy and Interrupt Routines . 2-5
2.2.4 Name Field 2-5
2.3 How to Create a Device Driver . . . 2-5
2.4 Installation of Device Drivers . . . 2-6
2.5 Request Header 2-6
2.5.1 Unit Code 2-7
2.5.2 Command Code Field 2-7
2.5.3 MEDIA CHECK and BUILD BPB 2-8
2.5.4 Status Word 2-9
2.6 Function Call Parameters 2-11
2.6.1 INIT 2-12
2.6.2 MEDIA CHECK 2-12
2.6.3 BUILD BPB 2-13
2.6.4 Media Descriptor Byte 2-15
2.6.5 READ OR WRITE 2-16
2.6.6 NON DESTRUCTIVE READ NO WAIT . . . 2-17
2.6.7 STATUS 2-18
2.6.8 FLUSH 2-18
2.7 The CLOCK Device 2-19
2.8 Example of Device Drivers 2-20
2.8.1 Block Device Driver 2-20
2.8.2 Character Device Driver 2-34

Chapter 3 MS-DOS Technical Information

3.1 MS-DOS Initialization 3-1
3.2 The Command Processor 3-1
3.3 MS-DOS Disk Allocation 3-3
3.4 MS-DOS Disk Directory 3-3
3.5 File Allocation Table 3-7
3.5.1 How to Use the File Allocation Table 3-8
3.6 IBM 5 1/4" MS-DOS Disk Formats . . . 3-9

Chapter 4 MS-DOS Control Blocks and Work Areas

4.1 Typical MS-DOS Memory Map 4-1
4.2 MS-DOS Program Segment 4-2

Chapter 5 EXE File Structure and Loading

Index

GENERAL INTRODUCTION

The Microsoft(R) MS(tm)-DOS Programmer's Reference Manual is a technical reference manual for system programmers. This manual contains a description and examples of all MS-DOS 2.0 system calls and interrupts (Chapter 1). Chapter 2, "MS-DOS 2.0 Device Drivers" contains information on how to install your own device drivers on MS-DOS. Two examples of device driver programs (one serial and one block) are included in Chapter 2. Chapters 3 through 5 contain technical information about MS-DOS, including MS-DOS disk allocation (Chapter 3), MS-DOS control blocks and work areas (Chapter 4), and EXE file structure and loading (Chapter 5).

CHAPTER 1

SYSTEM CALLS

1.1 INTRODUCTION

MS-DOS provides two types of system calls: interrupts and function requests. This chapter describes the environments from which these routines can be called, how to call them, and the processing performed by each.

1.2 PROGRAMMING CONSIDERATIONS

The system calls mean you don't have to invent your own ways to perform these primitive functions, and make it easier to write machine-independent programs.

1.2.1 Calling From Macro Assembler

The system calls can be invoked from Macro Assembler simply by moving any required data into registers and issuing an interrupt. Some of the calls destroy registers, so you may have to save registers before using a system call. The system calls can be used in macros and procedures to make your programs more readable; this technique is used to show examples of the calls.

1.2.2 Calling From A High-Level Language

The system calls can be invoked from any high-level language whose modules can be linked with assembly-language modules.

Calling from Microsoft Basic: Different techniques are used to invoke system calls from the compiler and interpreter. Compiled modules can be linked with assembly-language modules; from the interpreter, the CALL statement or USER function can be used to execute the appropriate 8086 object code.

<u>Calling from Microsoft Pascal</u>: In addition to linking with an assembly-language module, Microsoft Pascal includes a function (DOSXQQ) that can be used directly from a Pascal program to call a function request.

<u>Calling from Microsoft FORTRAN</u>: Modules compiled with Microsoft FORTRAN can be linked with assembly-language modules.

1.2.3 Returning Control To MS-DOS

Control can be returned to MS-DOS in any of four ways:

1. Call Function Request 4CH

   ```
   MOV  AH,4CH
   INT  21H
   ```

   This is the preferred method.

2. Call Interrupt 20H:

   ```
   INT  20H
   ```

3. Jump to location 0 (the beginning of the Program Segment Prefix):

   ```
   JMP  0
   ```

   Location 0 of the Program Segment Prefix contains an INT 20H instruction, so this technique is simply one step removed from the first.

4. Call Function Request 00H:

   ```
   MOV   AH,00H
   INT   21H
   ```

   This causes a jump to location 0, so it is simply one step removed from technique 2, or two steps removed from technique 1.

1.2.4 Console And Printer Input/Output Calls

The console and printer system calls let you read from and write to the console device and print on the printer without using any machine-specific codes. You can still take advantage of specific capabilities (display attributes such as positioning the cursor or erasing the screen, printer attributes such as double-strike or underline, etc.) by using constants for these codes and reassembling once with the correct constant values for the attributes.

1.2.5 Disk I/O System Calls

Many of the system calls that perform disk input and output require placing values into or reading values from two system control blocks: the File Control Block (FCB) and directory entry.

1.3 FILE CONTROL BLOCK (FCB)

The Program Segment Prefix includes room for two FCBs at offsets 5CH and 6CH. The system call descriptions refer to unopened and opened FCBs. An unopened FCB is one that contains only a drive specifier and filename, which can contain wild card characters (* and ?). An opened FCB contains all fields filled by the Open File system call (Function 0FH). Table 1.1 describes the fields of the FCB.

Table 1.1 Fields of File Control Block (FCB)

Name	Size (bytes)	Offset Hex	Offset Decimal
Drive number	1	00H	0
Filename	8	01-08H	1-8
Extension	3	09-0BH	9-11
Current block	2	0CH,0DH	12,13
Record size	2	0EH,0FH	14,15
File size	4	10-13H	16-19
Date of last write	2	14H,15H	20,21
Time of last write	2	16H,17H	22,23
Reserved	8	18-1FH	24-31
Current record	1	20H	32
Relative record	4	21-24H	33-36

1.3.1 Fields Of The FCB

Drive Number (offset 00H): Specifies the disk drive; 1 means drive A: and 2 means drive B:. If the FCB is to be used to create or open a file, this field can be set to 0 to specify the default drive; the Open File system call Function (0FH) sets the field to the number of the default drive.

Filename (offset 01H): Eight characters, left-aligned and padded (if necessary) with blanks. If you specify a reserved device name (such as LPT1), do not put a colon at the end.

Extension (offset 09H): Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).

Current Block (offset 0CH): Points to the block (group of 128 records) that contains the current record. This field and the Current Record field (offset 20H) make up the record pointer. This field is set to 0 by the Open File system call.

Record Size (offset 0EH): The size of a logical record, in bytes. Set to 128 by the Open File system call. If the record size is not 128 bytes, you must set this field after opening the file.

File Size (offset 10H): The size of the file, in bytes. The first word of this 4-byte field is the low-order part of the size.

Date of Last Write (offset 14H): The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 15H
| Y | Y | Y | Y | Y | Y | Y | M |
 15                        9 8
```

```
Offset 14H
| M | M | M | D | D | D | D | D |
           5 4                0
```

Time of Last Write (offset 16H): The time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

```
Offset 17H
| H | H | H | H | H | M | M | M |
 15               11 10
```

```
Offset 16H
| M | M | M | S | S | S | S | S |
           5 4                0
```

Reserved (offset 18H): These fields are reserved for use by MS-DOS.

Current Record (offset 20H): Points to one of the 128 records in the current block. This field and the Current Block field (offset 0CH) make up the record pointer. This field is not initialized by the Open File system call. You must set it before doing a sequential read or write to the file.

Relative Record (offset 21H): Points to the currently selected record, counting from the beginning of the file (starting with 0). This field is not initialized by the Open File system call. You must set it before doing a random read or write to the file. If the record size is less than 64 bytes, both words of this field are used; if the record size is 64 bytes or more, only the first three bytes are used.

NOTE

> If you use the FCB at offset 5CH of the Program Segment Prefix, the last byte of the Relative Record field is the first byte of the unformatted parameter area that starts at offset 80H. This is the default Disk Transfer Address.

1.3.2 Extended FCB

The Extended File Control Block is used to create or search for directory entries of files with special attributes. It adds the following 7-byte prefix to the FCB:

Name	Size (bytes)	Offset (Decimal)
Flag byte (255, or FFH)	1	-7
Reserved	5	-6
Attribute byte: 02H = Hidden file 04H = System file	1	-1

1.3.3 Directory Entry

A directory contains one entry for each file on the disk. Each entry is 32 bytes; Table 1.2 describes the fields of an entry.

Table 1.2 Fields of Directory Entry

Name	Size (bytes)	Offset Hex	Offset Decimal
Filename	8	00-07H	0-7
Extension	3	08-0AH	8-10
Attributes	1	0BH	11
Reserved	10	0C-15H	12-21

Time of last write	2	16H,17H	22,23
Date of last read	2	18H,19H	24,25
Reserved	2	1AH,1BH	26,27
File size	4	1C-1FH	28-31

1.3.4 Fields Of The FCB

Filename (offset 00H): Eight characters, left-aligned and padded (if necessary) with blanks. MS-DOS uses the first byte of this field for two special codes:

00H (0)	End of allocated directory
E5H (229)	Free directory entry

Extension (offset 08H): Three characters, left-aligned and padded (if necessary) with blanks. This field can be all blanks (no extension).

Attributes (offset 0BH): Attributes of the file:

Value			
Hex	Binary	Dec	Meaning
01H	0000 0001	1	Read-only
02H	0000 0010	2	Hidden
04H	0000 0100	4	System
07H	0000 0111	7	Changeable with CHGMOD
08H	0000 1000	8	Volume-ID
0AH	0001 0000	10	Directory
16H	0001 0110	22	Hard attributes for FINDENTRY
20H	0020 0000	32	Archive

Reserved (offset 0CH): Reserved for MS-DOS.

Time of Last Write (offset 16H): The time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

```
Offset 17H
| H | H | H | H | H | M | M | M |
 15                11 10


Offset 16H
| M | M | M | S | S | S | S | S |
          5 4                 0
```

Date of Last Write (offset 18H): The date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 19H
| Y | Y | Y | Y | Y | Y | Y | M |
 15                         9 8

Offset 18H
| M | M | M | D | D | D | D | D |
           5 4                0
```

File Size (offset 1CH): The size of the file, in bytes. The first word of this 4-byte field is the low-order part of the size.

1.4 SYSTEM CALL DESCRIPTIONS

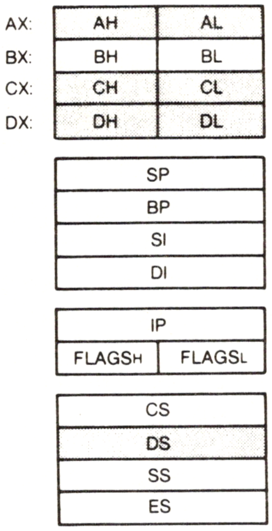
Many system calls require that parameters be loaded into one or more registers before the call is issued; most calls return information in the registers (usually a code that describes the success or failure of the operation). The description of system calls 00H-2EH includes the following:

> A drawing of the 8088 registers that shows their contents before and after the system call.
>
> A more complete description of the register contents required before the system call.
>
> A description of the processing performed.
>
> A more complete description of the register contents after the system call.
>
> An example of its use.

The description of system calls 2FH-57H includes the following:

> A drawing of the 8088 registers that shows their contents before and after the system call.
>
> A more complete description of the register contents required before the system call.
>
> A description of the processing performed.
>
> Error returns from the system call.
>
> An example of its use.

Figure 1 is an example of how each system call is described. Function 27H, Random Block Read, is shown.



Call
AH = 27H
DS:DX
  Opened FCB
CX
  Number of blocks to read

Return
AL
  0 = Read completed successfully
  1 = EOF
  2 = End of segment
  3 = EOF, partial record
CX
  Number of blocks read

Figure 1. Example of System Call Description

1.4.1 Programming Examples

A macro is defined for each system call, then used in some examples. In addition, a few other macros are defined for use in the examples. The use of macros allows the examples to be more complete programs, rather than isolated uses of the system calls. All macro definitions are listed at the end of the chapter.

The examples are not intended to represent good programming practice. In particular, error checking and good human interface design have been sacrificed to conserve space. You may, however, find the macros a convenient way to include system calls in your assembly language programs.

A detailed description of each system call follows. They are listed in numeric order; the interrupts are described first, then the function requests.

NOTE

> Unless otherwise stated, all numbers in the system call descriptions -- both text and code -- are in hex.

1.5 XENIX COMPATIBLE CALLS

MS-DOS 2.0 supports hierarchical (i.e., tree-structured) directories, similar to those found in the Xenix operating system. (For information on tree-structured directories, refer to the MS-DOS User's Guide.)

The following system calls are compatible with the Xenix system:

Function 39H	Create Sub-Directory
Function 3AH	Remove a Directory Entry
Function 3BH	Change the Current Directory
Function 3CH	Create a File
Function 3DH	Open a File
Function 3FH	Read From File/Device
Function 40H	Write to a File or Device
Function 41H	Delete a Directory Entry
Function 42H	Move a File Pointer
Function 43H	Change Attributes
Function 44H	I/O Control for Devices
Function 45H	Duplicate a File Handle
Function 46H	Force a Duplicate of a Handle
Function 4BH	Load and Execute a Program
Function 4CH	Terminate a Process
Function 4DH	Retrieve the Return Code of a Child

There is no restriction in MS-DOS 2.0 on the depth of a tree (the length of the longest path from root to leaf) except in the number of allocation units available. The root directory will have a fixed number of entries (64 for the single-sided disk). For non-root directories, the number of files per directory is only limited by the number of allocation units available.

Pre-2.0 disks will appear to MS-DOS 2.0 as having only a root directory with files in it and no subdirectories.

Implementation of the tree structure is simple. The root directory is the pre-2.0 directory. Subdirectories of the root have a special attribute set indicating that they are directories. The subdirectories themselves are files, linked through the FAT as usual. Their contents are identical in character to the contents of the root directory.

Pre-2.0 programs that use system calls not described in this chapter will be unable to make use of files in other directories. Those files not necessary for the current task will be placed in other directories.

Attributes apply to the tree-structured directories in the following manner:

Attribute	Meaning/Function for files	Meaning/Function for directories
volume_id	Present at the root. Only one file may have this set.	Meaningless.
directory	Meaningless.	Indicates that the directory entry is a directory. Cannot be changed with 43H.
read_only	Old fcb-create, new Create, new open (for write or read/write) will fail.	Meaningless.
archive	Set when file is written. Set/reset via Function 43H.	Meaningless.
hidden/ system	Prevents file from being found in search first/search next. Old open will fail.	Prevents directory entry from being found. Function 3BH will still work.

1.6 INTERRUPTS

MS-DOS reserves interrupts 20H through 3FH for its own use. The table of interrupt routine addresses (vectors) is maintained in locations 80H-FCH. Table 1.3 lists the interrupts in numeric order; Table 1.4 lists the interrupts in alphabetic order (of the description). User programs should only issue Interrupts 20H, 21H, 25H, 26H, and 27H. (Function Requests 4CH and 31H are the preferred method for Interrupts 20H and 27H for versions of MS-DOS that are 2.0 and higher.)

NOTE

Interrupts 22H, 23H, and 24H are not interrupts that can be issued by user programs; they are simply locations where a segment and offset address are stored.

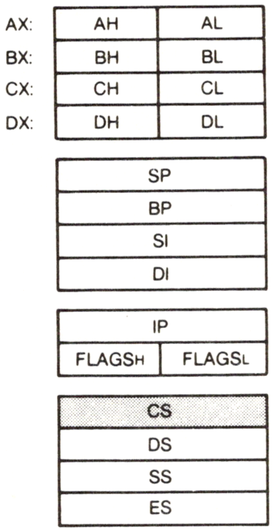
Table 1.3 MS-DOS Interrupts, Numeric Order

Interrupt Hex	Dec	Description
20H	32	Program Terminate
21H	33	Function Request
22H	34	Terminate Address
23H	35	<CTRL-C> Exit Address
24H	36	Fatal Error Abort Address
25H	37	Absolute Disk Read
26H	38	Absolute Disk Write
27H	39	Terminate But Stay Resident
28-40H	40-64	RESERVED -- DO NOT USE

Table 1.4 MS-DOS Interrupts, Alphabetic Order

Description	Interrupt Hex	Dec
Absolute Disk Read	25H	37
Absolute Disk Write	26H	38
<CTRL-C> Exit Address	23H	35
Fatal Error Abort Address	24H	36
Function Request	21H	33
Program Terminate	20H	32
RESERVED -- DO NOT USE	28-40H	40-64
Terminate Address	22H	34
Terminate But Stay Resident	27H	39

Program Terminate (Interrupt 20H)



Call

CS

Segment address of Program Segment Prefix

Return

None

Interrupt 20H causes the current process to terminate and returns control to its parent process. All open file handles are closed and the disk cache is cleaned. This interrupt is almost always is used in old .COM files for termination.

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt.

The following exit addresses are restored from the Program Segment Prefix:

Exit Address	Offset
Program Terminate	0AH
CONTROL-C	0EH
Critical Error	12H

All file buffers are flushed to disk.

NOTE

Close all files that have changed in length before issuing this interrupt. If a changed file is not closed, its length is not recorded correctly in the directory. See Functions 10H and 3EH for a description of the Close File system calls.

Interrupt 20H is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function Request 4CH, Terminate a Process.

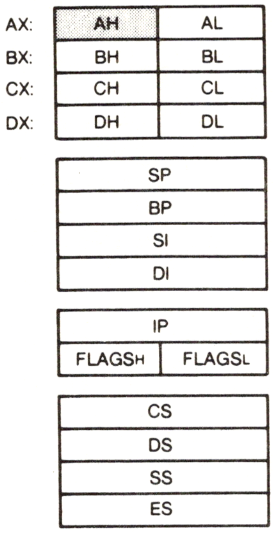
Macro Definition: terminate macro

```
terminate macro
          int 20H
          endm
```

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
   INT 20H
;There is no return from this interrupt
```

Function Request (Interrupt 21H)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH
  Function number
Other registers as specified in
  individual function

Return
As specified in individual function

The AH register must contain the number of the system function. See Section 1.7, "Function Requests," for a description of the MS-DOS system functions.

NOTE

No macro is defined for this interrupt, because all function descriptions in this chapter that define a macro include Interrupt 21H.

Example

To call the Get Time function:

```
mov   ah,2CH        ;Get Time is Function 2CH
int   21H           ;THIS INTERRUPT
```

Terminate Address (Interrupt 22H)
CONTROL-C Exit Address (Interrupt 23H)
Fatal Error Abort Address (Interrupt 24H)

These are not true interrupts, but rather storage locations for a segment and offset address. The interrupts are issued by MS-DOS under the specified circumstance. You can change any of these addresses with Function Request 25H (Set Vector) if you prefer to write your own interrupt handlers.

Interrupt 22H -- Terminate Address

When a program terminates, control transfers to the address at offset 0AH of the Program Segment Prefix. This address is copied into the Program Segment Prefix, from the Interrupt 22H vector, when the segment is created.

Interrupt 23H -- CONTROL-C Exit Address

If the user types CONTROL-C during keyboard input or display output, control transfers to the INT 23H vector in the interrupt table. This address is copied into the Program Segment Prefix, from the Interrupt 23H vector, when the segment is created.

If the CONTROL-C routine preserves all registers, it can end with an IRET instruction (return from interrupt) to continue program execution. When the interrupt occurs, all registers are set to the value they had when the original call to MS-DOS was made. There are no restrictions on what a CONTROL-C handler can do -- including MS-DOS function calls -- so long as the registers are unchanged if IRET is used.

If Function 09H or 0AH (Display String or Buffered Keyboard Input) is interrupted by CONTROL-C, the three-byte sequence 03H-0DH-0AH (ETX-CR-LF) is sent to the display and the function resumes at the beginning of the next line.

If the program creates a new segment and loads a second program that changes the CONTROL-C address, termination of the second program restores the CONTROL-C address to its value before execution of the second program.

Interrupt 24H -- Fatal Error Abort Address

If a fatal disk error occurs during execution of one of the disk I/O function calls, control transfers to the INT 24H vector in the vector table. This address is copied into the Program Segment Prefix, from the Interrupt 24H vector, when the segment is created.

BP:SI contains the address of a Device Header Control Block from which additional information can be retrieved.

NOTE

> Interrupt 24H is not issued if the failure occurs during execution of Interrupt 25H (Absolute Disk Read) or Interrupt 26H (Absolute Disk Write). These errors are usually handled by the MS-DOS error routine in COMMAND.COM that retries the disk operation, then gives the user the choice of aborting, retrying the operation, or ignoring the error. The following topics give you the information you need about interpreting the error codes, managing the registers and stack, and controlling the system's response to the error in order to write your own error-handling routines.

Error Codes

When an error-handling program gains control from Interrupt 24H, the AX and DI registers can contain codes that describe the error. If Bit 7 of AH is 1, the error is either a bad image of the File Allocation Table or an error occurred on a character device. The device header passed in BP:SI can be examined to determine which case exists. If the attribute byte high order bit indicates a block device, then the error was a bad FAT. Otherwise, the error is on a character device.

The following are error codes for Interrupt 24H:

Error Code	Description
0	Attempt to write on write-protected disk
1	Unknown unit
2	Drive not ready
3	Unknown command
4	Data error
5	Bad request structure length
6	Seek error
7	Unknown media type
8	Sector not found
9	Printer out of paper
A	Write fault
B	Read fault
C	General failure

The user stack will be in effect (the first item described below is at the top of the stack), and will contain the following from top to bottom:

```
IP      MS-DOS registers from
CS      issuing INT 24H
FLAGS

AX      User registers at time of original
BX      INT 21H request
CX
DX
SI
DI
BP
DS
ES

IP      From the original INT 21H
CS      from the user to MS-DOS
FLAGS
```

The registers are set such that if an IRET is executed, MS-DOS will respond according to (AL) as follows:

```
(AL)=0   ignore the error
    =1   retry the operation
    =2   terminate the program via INT 23H
```

Notes:

1. Before giving this routine control for disk errors, MS-DOS performs five retries.

2. For disk errors, this exit is taken only for errors occurring during an Interrupt 21H. It is not used for errors during Interrupts 25H or 26H.

3. This routine is entered in a disabled state.

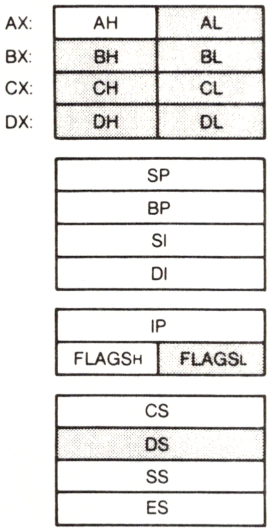
4. The SS, SP, DS, ES, BX, CX, and DX registers must be preserved.

5. This interrupt handler should refrain from using MS-DOS funtion calls. If necessary, it may use calls 01H through 0CH. Use of any other call will destroy the MS-DOS stack and will leave MS-DOS in an unpredictable state.

6. The interrupt handler must not change the contents of the device header.

7. If the interrupt handler will handle errors rather than returning to MS-DOS, it should restore the application program's registers from the stack, remove all but the last three words on the stack, then issue an IRET. This will return to the program immediately after the INT 21H that experienced the error. Note that if this is done, MS-DOS will be in an unstable state until a function call higher than 0CH is issued.

Absolute Disk Read (Interrupt 25H)



Call

AL
  Drive number
DS:BX
  Disk Transfer Address
CX
  Number of sectors
DX
  Beginning relative sector

Return

AL
  Error code if CF=1
FlagsL
  CF = 0 if successful
     = 1 if not successful

The registers must contain the following:

AL	Drive number (0=A, 1=B, etc.).
BX	Offset of Disk Transfer Address (from segment address in DS).
CX	Number of sectors to read.
DX	Beginning relative sector.

This interrupt transfers control to the MS-DOS BIOS. The number of sectors specified in CX is read from the disk to the Disk Transfer Address. Its requirements and processing are identical to Interrupt 26H, except data is read rather than written.

NOTE

> All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags.) Be sure to pop the stack upon return to prevent uncontrolled growth.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H earlier in this section for the codes and their meaning).

Macro Definition:

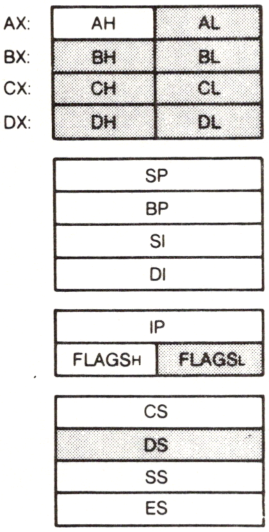
```
abs_disk_read  macro  disk,buffer,num_sectors,start
               mov    al,disk
               mov    bx,offset buffer
               mov    cx,num_sectors
               mov    dh,start
               int    25H
               endm
```

Example

The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:. It uses a buffer of 32K bytes:

```
prompt     db   "Source in A, target in B",13,10
           db   "Any key to start. $"
start      dw    0
buffer     db    64 dup (512 dup (?))  ;64 sectors
           .
           .
int_25H:   display prompt          ;see Function 09H
           read_kbd                ;see Function 08H
           mov     cx,5            ;copy 5 groups of
                                     ;64 sectors
copy:      push    cx              ;save the loop counter
           abs_disk_read  0,buffer,64,start  ;THIS INTERRUPT
           abs_disk_write 1,buffer,64,start  ;see INT 26H
           add  start,64           ;do the next 64 sectors
           pop  cx                 ;restore the loop counter
           loop copy
```

Absolute Disk Write (Interrupt 26H)



Call
AL
  Drive number
DS:BX
  Disk Transfer Address
CX
  Number of sectors
DX
  Beginning relative sector

Return
AL
  Error code if CF = 1
FLAGSL
  CF = 0 if successful
       1 if not successful

The registers must contain the following:

AL	Drive number (0=A, 1=B, etc.).
BX	Offset of Disk Transfer Address (from segment address in DS).
CX	Number of sectors to write.
DX	Beginning relative sector.

This interrupt transfers control to the MS-DOS BIOS. The number of sectors specified in CX is written from the Disk Transfer Address to the disk. Its requirements and processing are identical to Interrupt 25H, except data is written to the disk rather than read from it.

NOTE

> All registers except the segment registers are destroyed by this call. Be sure to save any registers your program uses before issuing the interrupt.

The system pushes the flags at the time of the call; they are still there upon return. (This is necessary because data is passed back in the flags.) Be sure to pop the stack upon return to prevent uncontrolled growth.

If the disk operation was successful, the Carry Flag (CF) is 0. If the disk operation was not successful, CF is 1 and AL contains the MS-DOS error code (see Interrupt 24H for the codes and their meaning).

Macro Definition:

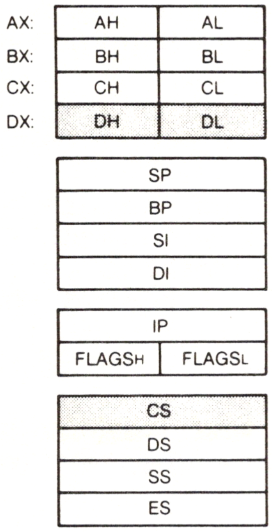
```
abs_disk_write  macro  disk,buffer,num_sectors,start
                mov    al,disk
                mov    bx,offset buffer
                mov    cx,num_sectors
                mov    dh,start
                int    26H
                endm
```

Example

The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:, verifying each write. It uses a buffer of 32K bytes:

```
off          equ   0
on           equ   1
             .
             .
prompt       db    "Source in A, target in B",13,10
             db    "Any key to start. $"
start        dw    0
buffer       db    64 dup (512 dup (?))  ;64 sectors
             .
             .
int_26H:     display prompt          ;see Function 09H
             read_kbd                ;see Function 08H
             verify  on              ;see Function 2EH
             mov     cx,5            ;copy 5 groups of 64 sectors
copy:        push    cx              ;save the loop counter
             abs_disk_read  0,buffer,64,start  ;see INT 25H
             abs_disk_write 1,buffer,64,start  ;THIS INTERRUPT
             add  start,64           ;do the next 64 sectors
             pop  cx                 ;restore the loop counter
             loop copy
             verify  off             ;see Function 2EH
```

Terminate But Stay Resident (Interrupt 27H)



Call
CS:DX
  First byte following
  last byte of code

Return
None

The Terminate But Stay Resident call is used to make a piece of code remain resident in the system after its termination. Typically, this call is used in .COM files to allow some device-specific interrupt handler to remain resident to process asynchronous interrupts.

DX must contain the offset (from the segment address in CS) of the first byte following the last byte of code in the program. When Interrupt 27H is executed, the program terminates but is treated as an extension of MS-DOS; it remains resident and is not overlaid by other programs when it terminates.

This interrupt is provided for compatibility with versions of MS-DOS prior to 2.0. New programs should use Function 31H, Keep Process.

Macro Definition:

```
stay_resident macro last_instruc
              mov   dx,offset last_instruc
              inc   dx
              int   27H
              endm
```

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
  mov     DX,LastAddress
  int    27H
;There is no return from this interrupt
```

1.7 FUNCTION REQUESTS

Most of the MS-DOS function calls require input to be passed to them in registers. After setting the proper register values, the function may be invoked in one of the following ways:

1. Place the function number in AH and execute a long call to offset 50H in your Program Segment Prefix. Note that programs using this method will not operate correctly on versions of MS-DOS that are lower than 2.0.

2. Place the function number in AH and issue Interrupt 21H. All of the examples in this chapter use this method.

3. An additional method exists for programs that were written with different calling conventions. This method should be avoided for all new programs. The function number is placed in the CL register and other registers are set according to the function specification. Then, an intrasegment call is made to location 5 in the current code segment. That location contains a long call to the MS-DOS function dispatcher. Register AX is always destroyed if this method is used; otherwise, it is the same as normal function calls. Note that this method is valid only for Function Requests 00H through 024H.

1.7.1 CP/M(R)-Compatible Calling Sequence

A different sequence can be used for programs that must conform to CP/M calling conventions:

1. Move any required data into the appropriate registers (just as in the standard sequence).

2. Move the function number into the CL register.

3. Execute an intrasegment call to location 5 in the current code segment.

This method can only be used with functions 00H through 24H that do not pass a parameter in AL. Register AX is always destroyed when a function is called in this manner.

1.7.2 Treatment Of Registers

When MS-DOS takes control after a function call, it switches to an internal stack. Registers not used to return information (except AX) are preserved. The calling program's stack must be large enough to accommodate the interrupt system -- at least 128 bytes in addition to other needs.

IMPORTANT NOTE

The macro definitions and extended example for MS-DOS system calls 00H through 2EH can be found at the end of this chapter.

Table 1.5 lists the function requests in numeric order; Table 1.6 list the function requests in alphabetic order (of the description).

Table 1.5 MS-DOS Function Requests, Numeric Order

Function Number	Function Name
00H	Terminate Program
01H	Read Keyboard and Echo
02H	Display Character
03H	Auxiliary Input
04H	Auxiliary Output
05H	Print Character
06H	Direct Console I/O
07H	Direct Console Input
08H	Read Keyboard
09H	Display String
0AH	Buffered Keyboard Input
0BH	Check Keyboard Status
0CH	Flush Buffer, Read Keyboard
0DH	Disk Reset
0EH	Select Disk
0FH	Open File
10H	Close File
11H	Search for First Entry
12H	Search for Next Entry
13H	Delete File
14H	Sequential Read
15H	Sequential Write
16H	Create File
17H	Rename File
19H	Current Disk
1AH	Set Disk Transfer Address

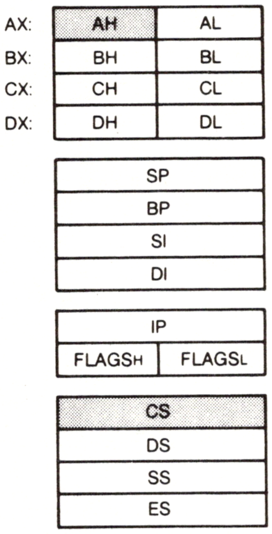
21H	Random Read
22H	Random Write
23H	File Size
24H	Set Relative Record
25H	Set Vector
27H	Random Block Read
28H	Random Block Write
29H	Parse File Name
2AH	Get Date
2BH	Set Date
2CH	Get Time
2DH	Set Time
2EH	Set/Reset Verify Flag
2FH	Get Disk Transfer Address
30H	Get DOS Version Number
31H	Keep Process
33H	CONTROL-C Check
35H	Get Interrupt Vector
36H	Get Disk Free Space
38H	Return Country-Dependent Information
39H	Create Sub-Directory
3AH	Remove a Directory Entry
3BH	Change Current Directory
3CH	Create a File
3DH	Open a File
3EH	Close a File Handle
3FH	Read From File/Device
40H	Write to a File/Device
41H	Delete a Directory Entry
42H	Move a File Pointer
43H	Change Attributes
44H	I/O Control for Devices
45H	Duplicate a File Handle
46H	Force a Duplicate of a Handle
47H	Return Text of Current Directory
48H	Allocate Memory
49H	Free Allocated Memory
4AH	Modify Allocated Memory Blocks
4BH	Load and Execute a Program
4CH	Terminate a Process
4DH	Retrieve the Return Code of a Child
4EH	Find Match File
4FH	Step Through a Directory Matching Files
54H	Return Current Setting of Verify
56H	Move a Directory Entry
57H	Get/Set Date/Time of File

Table 1.6 MS-DOS Function Requests, Alphabetic Order

Function Name	Number
Allocate Memory	48H
Auxiliary Input	03H
Auxiliary Output	04H
Buffered Keyboard Input	0AH
Change Attributes	43H
Change the Current Directory	3BH
Check Keyboard Status	0BH
Close a File Handle	3EH
Close File	10H
CONTROL-C Check	33H
Create a File	3CH
Create File	16H
Create Sub-Directory	39H
Current Disk	19H
Delete a Directory Entry	41H
Delete File	13H
Direct Console Input	07H
Direct Console I/O	06H
Disk Reset	0DH
Display Character	02H
Display String	09H
Duplicate a File Handle	45H
File Size	23H
Find Match File	4EH
Flush Buffer, Read Keyboard	0CH
Force a Duplicate of a Handle	46H
Free Allocated Memory	49H
Get Date	2AH
Get Disk Free Space	36H
Get Disk Transfer Address	2FH
Get DOS Version Number	30H
Get Interrupt Vector	35H
Get Time	2CH
Get/Set Date/Time of File	57H
I/O Control for Devices	44H
Keep Process	31H
Load and Execute a Program	4BH
Modify Allocated Memory Blocks	4AH
Move a Directory Entry	56H
Move a File Pointer	42H
Open a File	3DH
Open File	0FH
Parse File Name	29H
Print Character	05H
Random Block Read	27H
Random Block Write	28H
Random Read	21H
Random Write	22H
Read From File/Device	3FH
Read Keyboard	08H
Read Keyboard and Echo	01H

Remove a Directory Entry	3AH
Rename File	17H
Retrieve the Return Code of a Child	4DH
Return Current Setting of Verify	54H
Return Country-Dependent Information	38H
Return Text of Current Directory	47H
Search for First Entry	11H
Search for Next Entry	12H
Select Disk	0EH
Sequential Read	14H
Sequential Write	15H
Set Date	2BH
Set Disk Transfer Address	1AH
Set Relative Record	24H
Set Time	2DH
Set Vector	25H
Set/Reset Verify Flag	2EH
Step Through a Directory Matching	4FH
Terminate a Process	4CH
Terminate Program	00H
Write to a File/Device	40H

Terminate Program (Function 00H)



Call
AH = 00H
CS
  Segment address of
  Program Segment Prefix

Return
None

Function 00H is called by Interrupt 20H; it performs the same processing.

The CS register must contain the segment address of the Program Segment Prefix before you call this interrupt.

The following exit addresses are restored from the specified offsets in the Program Segment Prefix:

Program terminate	0AH
CONTROL-C	0EH
Critical error	12H

All file buffers are flushed to disk.

Warning: Close all files that have changed in length before calling this function. If a changed file is not closed, its length is not recorded correctly in the directory. See Function 10H for a description of the Close File system call.

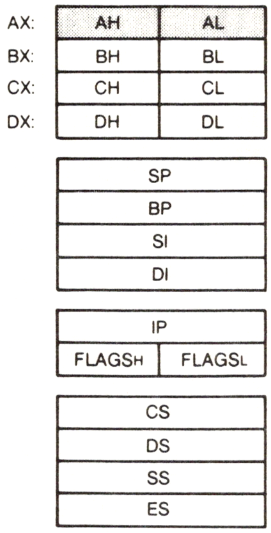
Macro Definition:

```
terminate_program  macro
                   xor     ah,ah
                   int     21H
                   endm
```

Example

```
;CS must be equal to PSP values given at program start
;(ES and DS values)
  mov   ah,0
  int   21H
;There are no returns from this interrupt
```

Read Keyboard and Echo (Function 01H)



Call
AH = 01H

Return
AL
  Character typed

Function 01H waits for a character to be typed at the keyboard, then echos the character to the display and returns it in AL. If the character is CONTROL-C, Interrupt 23H is executed.

Macro Definition:

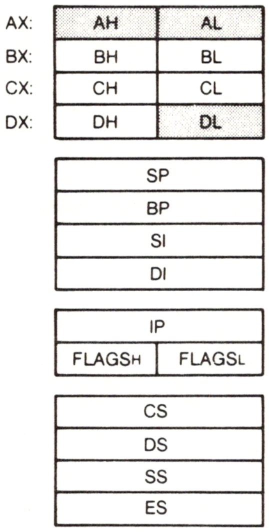
```
read_kbd_and_echo  macro
                   mov  ah, 01H
                   int  21H
                   endm
```

Example

The following program both displays and prints characters as they are typed. If RETURN is pressed, the program sends Line Feed-Carriage Return to both the display and the printer:

```
func_01H:  read_kbd_and_echo           ;THIS FUNCTION
           print_char   al             ;see Function 05H
           cmp          al,0DH         ;is it a CR?
           jne          func_01H       ;no, print it
           print_char   10             ;see Function 05H
           display_char 10             ;see Function 02H
           jmp          func_01H       ;get another character
```

Display Character (Function 02H)



Call
AH = 02H
DL
  Character to be displayed

Return
None

Function 02H displays the character in DL. If CONTROL-C is typed, Interrupt 23H is issued.

Macro Definition:

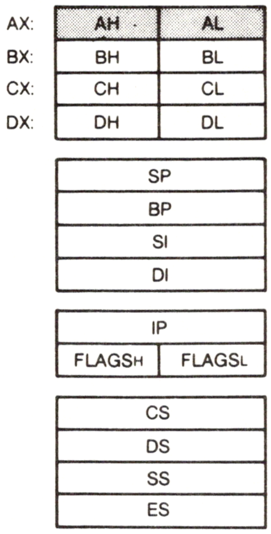
```
display_char macro   character
             mov     dl,character
             mov     ah,02H
             int     21H
             endm
```

Example

The following program converts lowercase characters to uppercase before displaying them:

```
func_02H:   read_kbd                  ;see Function 08H
            cmp     al,"a"
            jl      uppercase         ;don't convert
            cmp     al,"z"
            jg      uppercase         ;don't convert
            sub     al,20H            ;convert to ASCII code
                                      ;for uppercase
uppercase:  display_char al           ;THIS FUNCTION
            jmp     func_02H:         ;get another character
```

Auxiliary Input (Function 03H)



Call
AH = 03H

Return
AL
  Character from auxiliary device

Function 03H waits for a character from the auxiliary input device, then returns the character in AL. This system call does not return a status or error code.

If a CONTROL-C has been typed at console input, Interrupt 23H is issued.

Macro Definition:

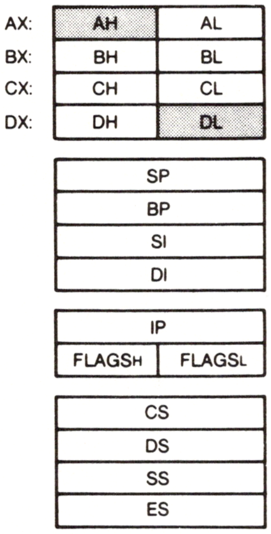
```
aux_input  macro
           mov   ah,03H
           int   21H
           endm
```

Example

The following program prints characters as they are received from the auxiliary device. It stops printing when an end-of-file character (ASCII 26, or CONTROL-Z) is received:

```
func_03H:  aux_input        ;THIS FUNCTION
           cmp   al,1AH     ;end of file?
           je    continue   ;yes, all done
           print_char  al   ;see Function 05H
           jmp   func_03H   ;get another character
continue:  .
           .
```

Auxiliary Output (Function 04H)



Call
AH = 04H
DL
  Character for auxiliary device

Return
None

Function 04H sends the character in DL to the auxiliary output device. This system call does not return a status or error code.

If a CONTROL-C has been typed at console input, Interrupt 23H is issued.

Macro Definition:

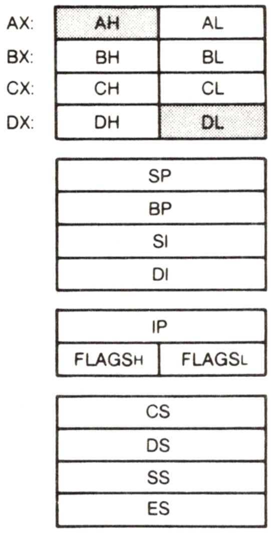
```
aux_output  macro  character
            mov    dl,character
            mov    ah,04H
            int    21H
            endm
```

Example

The following program gets a series of strings of up to 80 bytes from the keyboard, sending each to the auxiliary device. It stops when a null string (CR only) is typed:

```
string    db    81 dup(?) ;see Function 0AH
          .
          .
func_04H:get_string  80,string       ;see Function 0AH
         cmp  string[1],0             ;null string?
         je   continue                ;yes, all done
         mov  cx, word ptr string[1] ;get string length
         mov  bx,0                    ;set index to 0
send_it: aux_output string[bx+2]      ;THIS FUNCTION
         inc  bx                      ;bump index
         loop  send_it                ;send another character
         jmp  func_04H                ;get another string
continue: .
          .
```

Print Character (Function 05H)



Call
AH = 05H
DL
  Character for printer

Return
None

Function 05H prints the character in DL on the standard printer device. If CONTROL-C has been typed at console input, Interrupt 23H is issued.

Macro Definition:

```
print_char macro character
           mov   dl,character
           mov   ah,05H
           int   21H
           endm
```

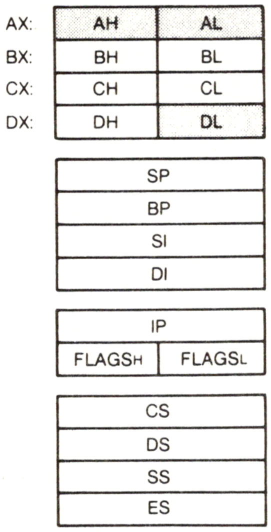
Example

The following program prints a walking test pattern on the printer. It stops if CONTROL-C is pressed.

```
line_num    db    0
            .
            .
func_05H:   mov   cx,60          ;print 60 lines
start_line: mov   bl,33          ;first printable ASCII
                                 ;character (!)
            add   bl,line_num    ;to offset one character
            push  cx             ;save number-of-lines counter
            mov   cx,80          ;loop counter for line
print_it:   print_char bl        ;THIS FUNCTION
            inc   bl             ;move to next ASCII character
            cmp   bl,126         ;last printable ASCII
                                 ;character (~)
            jl    no_reset       ;not there yet
            mov   bl,33          ;start over with (!)
```

```
no_reset:   loop  print_it      ;print another character
            print_char 13       ;carriage return
            print_char 10       ;line feed
            inc   line_num      ;to offset 1st char. of line
            pop   cx            ;restore #-of-lines counter
            loop  start_line;   ;print another line
```

Direct Console I/O (Function 06H)



Call
AH = 06H
DL
  See text

Return
AL
  If DL = FFH (255) before call, then Zero flag set means AL has character from keyboard.
  Zero flag not set means there was not a character to get, and AL = 0

The processing depends on the value in DL when the function is called:

> DL is FFH (255) -- If a character has been typed at the keyboard, it is returned in AL and the Zero flag is 0; if a character has not been typed, the Zero flag is 1.
>
> DL is not FFH -- The character in DL is displayed.

This function does not check for CONTROL-C.

Macro Definition:

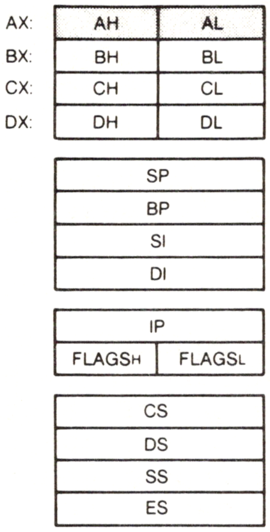
```
dir_console_io   macro switch
                 mov   dl,switch
                 mov   ah,06H
                 int   21H
                 endm
```

Example

The following program sets the system clock to 0 and continuously displays the time. When any character is typed, the display stops changing; when any character is typed again, the clock is reset to 0 and the display starts again:

```
time           db  "00:00:00.00",13,10,"$"  ;see Function 09H
;                                          ;for explanation of $
ten            db  10
               .
               .
func_06H:      set_time  0,0,0,0          ;see Function 2DH
read_clock:    get_time                   ;see Function 2CH
               convert  ch,ten,time       ;see end of chapter
               convert  cl,ten,time[3]    ;see end of chapter
               convert  dh,ten,time[6]    ;see end of chapter
               convert  dl,ten,time[9]    ;see end of chapter
               display  time              ;see Function 09H
               dir_console_io  FFH        ;THIS FUNCTION
               jne      stop              ;yes, stop timer
               jmp      read_clock        ;no, keep timer
                                          ;running
stop:          read_kbd                   ;see Function 08H
               jmp      func_06H          ;start over
```

Direct Console Input (Function 07H)



Call
AH = 07H

Return
AL
  Character from keyboard

Function 07H waits for a character to be typed, then returns it in AL. This function does not echo the character or check for CONTROL-C. (For a keyboard input function that echoes or checks for CONTROL-C, see Functions 01H or 08H.)

Macro Definition:

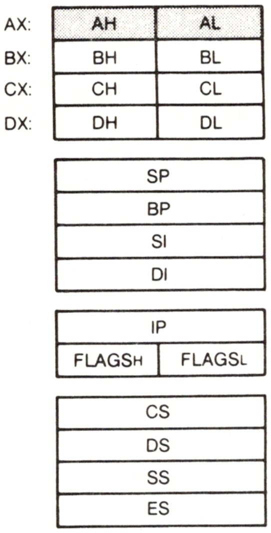
```
dir_console_input  macro
                   mov   ah,07H
                   int   21H
                   endm
```

Example

The following program prompts for a password (8 characters maximum) and places the characters into a string without echoing them:

```
password   db  8 dup(?)
prompt     db "Password: $"       ;see Function 09H for
                                  ;explanation of $
           .
           .
func_07H: display prompt          ;see Function 09H
          mov  cx,8               ;maximum length of password
          xor  bx,bx              ;so BL can be used as index
get_pass: dir_console_input       ;THIS FUNCTION
          cmp  al,0DH             ;was it a CR?
          je   continue           ;yes, all done
          mov  password bx ,al    ;no, put character in string
          inc  bx                 ;bump index
          loop get_pass           ;get another character
continue: .                       ;BX has length of password+1
          .
```

Read Keyboard (Function 08H)



Call
AH = 08H

Return
AL
Character from keyboard

Function 08H waits for a character to be typed, then returns it in AL. If CONTROL-C is pressed, Interrupt 23H is executed. This function does not echo the character. (For a keyboard input function that echoes the character or checks for CONTROL-C, see Function 01H.)

Macro Definition:

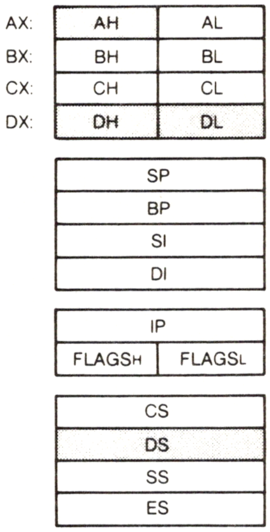
```
read_kbd  macro
          mov    ah,08H
          int    21H
          endm
```

Example

The following program prompts for a password (8 characters maximum) and places the characters into a string without echoing them:

```
password  db  8 dup(?)
prompt    db "Password: $"       ;see Function 09H
                                 ;for explanation of $

          .

          .
func_08H: display prompt         ;see Function 09H
          mov  cx,8              ;maximum length of password
          xor  bx,bx             ;BL can be an index
get_pass: read_kbd               ;THIS FUNCTION
          cmp  al,0DH            ;was it a CR?
          je   continue          ;yes, all done
          mov  password]bx[,al   ;no, put char. in string
          inc  bx                ;bump index
          loop get_pass          ;get another character
continue: .                      ;BX has length of password+1
          .
```

Display String (Function 09H)



Call
AH = 09H
DS:DX
  String to be displayed

Return
None

DX must contain the offset (from the segment address in DS) of a string that ends with "$". The string is displayed (the $ is not displayed).

Macro Definition:

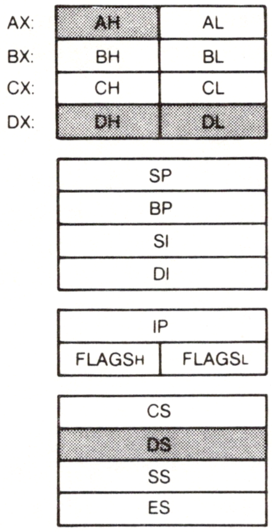
```
display  macro string
         mov   dx,offset string
         mov   ah,09H
         int   21H
         endm
```

Example

The following program displays the hexadecimal code of the key that is typed:

```
table    db     "0123456789ABCDEF"
sixteen  db      16
result   db     " - 00H",13,10,"$"  ;see text for
                                     ;explanation of $

         .
         .
func_09H:read_kbd_and_echo            ;see Function 01H
         convert al,sixteen,result]3[ ;see end of chapter
         display result               ;THIS FUNCTION
         jmp     func_09H             ;do it again
```

Buffered Keyboard Input (Function 0AH)



Call
AH = 0AH
DS:DX
  Input buffer

Return
None

DX must contain the offset (from the segment address in DS) of an input buffer of the following form:

Byte	Contents
1	Maximum number of characters in buffer, including the CR (you must set this value).
2	Actual number of characters typed, not counting the CR (the function sets this value).
3-n	Buffer; must be at least as long as the number in byte 1.

This function waits for characters to be typed. Characters are read from the keyboard and placed in the buffer beginning at the third byte until RETURN is typed. If the buffer fills to one less than the maximum, additional characters typed are ignored and ASCII 7 (BEL) is sent to the display until RETURN is pressed. The string can be edited as it is being entered. If CONTROL-C is typed, Interrupt 23H is issued.

The second byte of the buffer is set to the number of characters entered (not counting the CR).

Macro Definition:

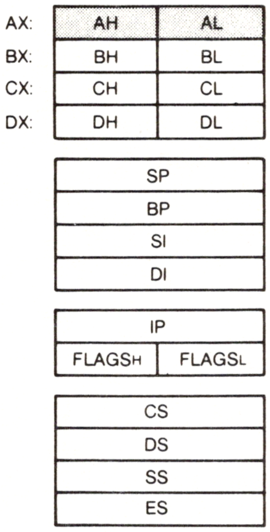
```
get_string  macro   limit,string
            mov     dx,offset string
            mov     string,limit
            mov     ah,0AH
            int     21H
            endm
```

Example

The following program gets a 16-byte (maximum) string from the keyboard and fills a 24-line by 80-character screen with it:

```
buffer            label byte
max_length        db    ?                 ;maximum length
chars_entered     db    ?                 ;number of chars.
string            db    17 dup (?)        ;16 chars + CR
strings_per_line dw     0                 ;how many strings
                                          ;fit on line
crlf              db    13,10,"$"
                  .
                  .
func_0AH:         get_string 17,buffer    ;THIS FUNCTION
                  xor   bx,bx             ;so byte can be
                                          ;used as index
                  mov   bl,chars_entered  ;get string length
                  mov   buffer[bx+2],"$"  ;see Function 09H
                  mov   al,50H            ;columns per line
                  cbw
                  div   chars_entered     ;times string fits
                                          ;on line
                  xor   ah,ah             ;clear remainder
                  mov   strings_per_line,ax ;save col. counter
                  mov   cx,24             ;row counter
display_screen:   push cx                 ;save it
                  mov   cx,strings_per_line ;get col. counter
display_line:     display  string         ;see Function 09H
                  loop display_line
                  display crlf            ;see Function 09H
                  pop   cx                ;get line counter
                  loop display_screen     ;display 1 more line
```

Check Keyboard Status (Function 0BH)



Call
AH = 0BH

Return
AL
255 (FFH) = characters in type-ahead buffer
0 = no characters in type-ahead buffer

Checks whether there are characters in the type-ahead buffer. If so, AL returns FFH (255); if not, AL returns 0. If CONTROL-C is in the buffer, Interrupt 23H is executed.

Macro Definition:

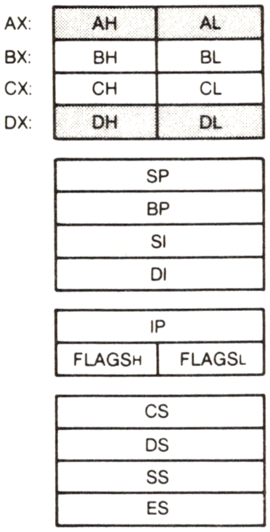
```
check_kbd_status  macro
                  mov    ah,0BH
                  int    21H
                  endm
```

Example

The following program continuously displays the time until any key is pressed.

```
time        db      "00:00:00.00",13,10,"$"
ten         db       10
            .
            .
func_0BH:   get_time                      ;see Function 2CH
            convert ch,ten,time           ;see end of chapter
            convert cl,ten,time[3]        ;see end of chapter
            convert dh,ten,time[6]        ;see end of chapter
            convert dl,ten,time[9]        ;see end of chapter
            display time                  ;see Function 09H
            check_kbd_status              ;THIS FUNCTION
            cmp     al,FFH                ;has a key been typed?
            je      all_done              ;yes, go home
            jmp     func_0BH              ;no, keep displaying
                                          ;time
```

Flush Buffer, Read Keyboard (Function 0CH)



Call
AH = 0CH
AL
  1, 6, 7, 8, or 0AH = The corresponding function is called.
  Any other value = no further processing.

Return
AL
  0 = Type-ahead buffer was flushed; no other processing performed.

The keyboard type-ahead buffer is emptied. Further processing depends on the value in AL when the function is called:

1, 6, 7, 8, or 0AH -- The corresponding MS-DOS function is executed.

Any other value -- No further processing; AL returns 0.

Macro Definition: flush_and_read_kbd

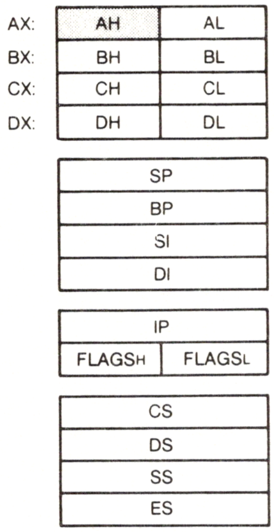
```
macro switch
mov   al,switch
mov   ah,0CH
int   21H
endm
```

Example

The following program both displays and prints characters as they are typed. If RETURN is pressed, the program sends Carriage Return-Line Feed to both the display and the printer.

```
func_0CH:  flush_and_read_kbd 1       ;THIS FUNCTION
           print_char   al            ;see Function 05H
           cmp          al,0DH        ;is it a CR?
           jne          func_0CH      ;no, print it
           print_char   10            ;see Function 05H
           display_char 10            ;see Function 02H
           jmp          func_0CH      ;get another character
```

Disk Reset (Function 0DH)



Call
AH = 0DH

Return
None

Function 0DH is used to ensure that the internal buffer cache matches the disks in the drives. This function writes out dirty buffers (buffers that have been modified), and marks all buffers in the internal cache as free.

Function 0DH flushes all file buffers. It does not update directory entries; you must close files that have changed to update their directory entries (see Function 10H, Close File). This function need not be called before a disk change if all files that changed were closed. It is generally used to force a known state of the system; CONTROL-C interrupt handlers should call this function.

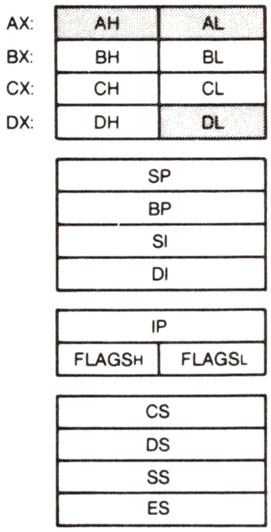
Macro Definition:

```
disk_reset macro disk
           mov   ah,0DH
           int   21H
           endm
```

Example

```
  mov     ah,0DH
  int     21H
;There are no errors returned by this call.
```

Select Disk (Function 0EH)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 0EH
DL
  Drive number
  (0 = A:, 1 = B:, etc.)

Return
AL
  Number of logical drives

The drive specified in DL (0 = A:, 1 = B:, etc.) is selected as the default disk. The number of drives is returned in AL.

Macro Definition:

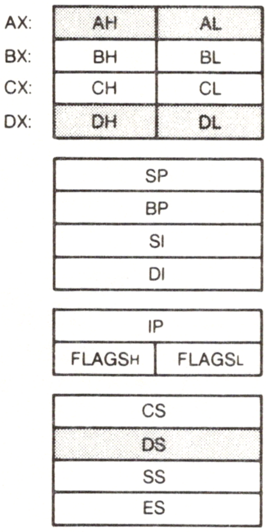
```
select_disk macro disk
            mov   dl,disk[-64]
            mov   ah,0EH
            int   21H
            endm
```

Example

The following program selects the drive not currently selected in a 2-drive system:

```
func_0EH: current_disk       ;see Function 19H
          cmp    al,00H      ;drive A: selected?
          je     select_b    ;yes, select B
          select_disk "A"    ;THIS FUNCTION
          jmp    continue
select_b: select_disk "B"    ;THIS FUNCTION
continue: .
          .
```

Open File (Function 0FH)



Call
AH = 0FH
DS:DX
  Unopened FCB

Return
AL
  0 = Directory entry found
  255 (FFH) = No directory entry found

DX must contain the offset (from the segment address in DS) of an unopened File Control Block (FCB). The disk directory is searched for the named file.

If a directory entry for the file is found, AL returns 0 and the FCB is filled as follows:

> If the drive code was 0 (default disk), it is changed to the actual disk used (1 = A:, 2 = B:, etc.). This lets you change the default disk without interfering with subsequent operations on this file.
>
> The Current Block field (offset 0CH) is set to zero.
>
> The Record Size (offset 0EH) is set to the system default of 128.
>
> The File Size (offset 10H), Date of Last Write (offset 14H), and Time of Last Write (offset 16H) are set from the directory entry.

Before performing a sequential disk operation on the file, you must set the Current Record field (offset 20H). Before performing a random disk operation on the file, you must set the Relative Record field (offset 21H). If the default record size (128 bytes) is not correct, set it to the correct length.

If a directory entry for the file is not found, AL returns FFH (255).

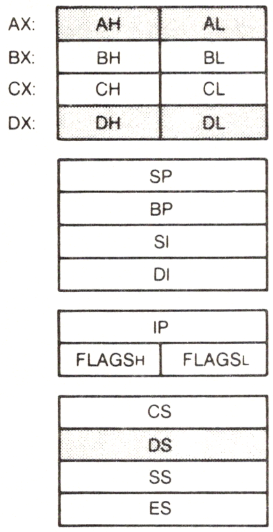
```
Macro Definition: open macro fcb
                       mov   dx,offset fcb
                       mov   ah,0FH
                       int   21H
                       endm
```

Example

The following program prints the file named TEXTFILE.ASC that is on the disk in drive B:. If a partial record is in the buffer at end-of-file, the routine that prints the partial record prints characters until it encounters an end-of-file mark (ASCII 26, or CONTROL-Z):

```
fcb            db    2,"TEXTFILEASC"
               db    25 dup (?)
buffer         db    128 dup (?)
               .
               .
func_0FH:      set_dta  buffer        ;see Function 1AH
               open  fcb              ;THIS FUNCTION
read_line:     read_seq  fcb          ;see Function 14H
               cmp   al,02H           ;end of file?
               je    all_done         ;yes, go home
               cmp   al,00H           ;more to come?
               jg    check_more       ;no, check for partial
                                      ;record
               mov   cx,128           ;yes, print the buffer
               xor   si,si            ;set index to 0
print_it:      print_char buffer[si]  ;see Function 05H
               inc   si               ;bump index
               loop  print_it         ;print next character
               jmp   read_line        ;read another record
check_more:    cmp   al,03H           ;part. record to print?
               jne   all_done         ;no
               mov   cx,128           ;yes, print it
               xor   si,si            ;set index to 0
find_eof:      cmp   buffer[si],26    ;end-of-file mark?
               je    all_done         ;yes
               print_char buffer[si]  ;see Function 05H
               inc   si               ;bump index to next
                                      ;character
               loop  find_eof
all_done:      close fcb              ;see Function 10H
```

Close File (Function 10H)



Call
AH = 10H
DS:DX
  Opened FCB

Return
AL
  0 = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (to the segment address in DS) of an opened FCB. The disk directory is searched for the file named in the FCB. This function must be called after a file is changed to update the directory entry.

If a directory entry for the file is found, the location of the file is compared with the corresponding entries in the FCB. The directory entry is updated, if necessary, to match the FCB, and AL returns 0.

If a directory entry for the file is not found, AL returns FFH (255).

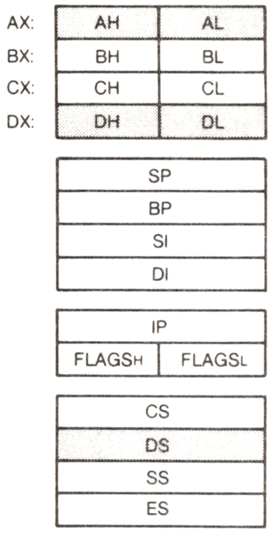
Macro Definition:
```
close macro fcb
      mov   dx,offset fcb
      mov   ah,10H
      int   21H
      endm
```

Example

The following program checks the first byte of the file named MOD1.BAS in drive B: to see if it is FFH, and prints a message if it is:

```
message        db   "Not saved in ASCII format",13,10,"$"
fcb            db     2,"MOD1    BAS"
               db     25  dup (?)
buffer         db     128 dup (?)
               .
               .
func_10H:      set_dta  buffer       ;see Function 1AH
               open  fcb             ;see Function 0FH
               read_seq  fcb         ;see Function 14H
               cmp    buffer,FFH        ;is first byte FFH?
               jne    all_done          ;no
               display  message         ;see Function 09H
all_done:      close fcb                ;THIS FUNCTION
```

Search for First Entry (Function 11H)



Call
AH = 11H
DS:DX
  Unopened FCB

Return
  0 = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an unopened FCB. The disk directory is searched for the first matching name. The name can have the ? wild card character to match any character. To search for hidden or system files, DX must point to the first byte of the extended FCB prefix.

If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address.

If a directory entry for the filename in the FCB is not found, AL returns FFH (255).

Notes:

If an extended FCB is used, the following search pattern is used:

1. If the FCB attribute is zero, only normal file entries are found. Entries for volume label, sub-directories, hidden, and system files will not be returned.

2. If the attribute field is set for hidden or system files, or directory entries, it is to be considered as an inclusive search. All normal file entries plus all entries matching the specified attributes are returned. To look at all directory entries except the volume label, the attribute byte may be set to hidden + system + directory (all 3 bits on).

3. If the attribute field is set for the volume label, it is considered an exclusive search, and only the volume label entry is returned.

Macro Definition:

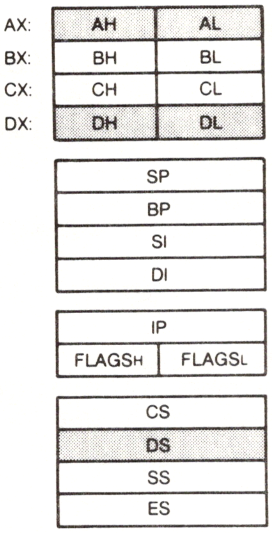
```
search_first macro fcb
             mov   dx,offset fcb
             mov   ah,11H
             int   21H
             endm
```

Example

The following program verifies the existence of a file named REPORT.ASM on the disk in drive B::

```
yes         db    "FILE EXISTS.$"
no          db    "FILE DOES NOT EXIST.$"
fcb         db     2,"REPORT  ASM"
            db     25 dup (?)
buffer      db     128 dup (?)
            .
            .
func_11H:   set_dta  buffer              ;see Function 1AH
            search_first fcb             ;THIS FUNCTION
            cmp      al,FFH              ;directory entry found?
            je       not_there           ;no
            display  yes                 ;see Function 09H
            jmp      continue
not_there:  display  no                  ;see Function 09H
continue:   display  crlf                ;see Function 09H
           .
           .
```

Search for Next Entry (Function 12H)



Call
AH = 12H
DS:DX
  Unopened FCB

Return
AL
  0 = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an FCB previously specified in a call to Function 11H. Function 12H is used after Function 11H (Search for First Entry) to find additional directory entries that match a filename that contains wild card characters. The disk directory is searched for the next matching name. The name can have the ? wild card character to match any character. To search for hidden or system files, DX must point to the first byte of the extended FCB prefix.

If a directory entry for the filename in the FCB is found, AL returns 0 and an unopened FCB of the same type (normal or extended) is created at the Disk Transfer Address.

If a directory entry for the filename in the FCB is not found, AL returns FFH (255).

Macro Definition:

```
search_next macro fcb
            mov   dx,offset fcb
            mov   ah,12H
            int   21H
            endm
```

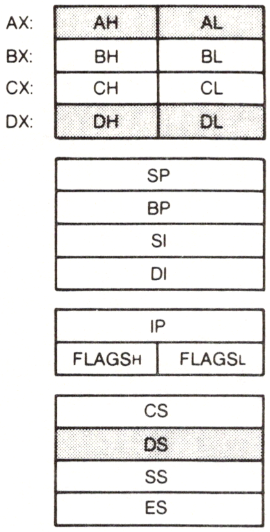
Example

The following program displays the number of files on the disk in drive B:

```
message     db    "No files",10,13,"$"
files       db     0
ten         db     10
fcb         db     2,"???????????"
            db     25  dup (?)
buffer      db     128 dup (?)
```

```
                .
                .
func_12H:    set_dta  buffer          ;see Function 1AH
             search_first  fcb        ;see Function 11H
             cmp  al,FFH              ;directory entry found?
             je   all_done            ;no, no files on disk
             inc  files               ;yes, increment file
                                      ;counter
search_dir: search_next  fcb          ;THIS FUNCTION
             cmp  al,FFH              ;directory entry found?
             je   done                ;no
             inc  files               ;yes, increment file
                                      ;counter
             jmp  search_dir          ;check again
done:        convert files,ten,message ;see end of chapter
all_done:    display  message         ;see Function 09H
```

Delete File (Function 13H)



Call
AH = 13H
DS:DX
  Unopened FCB

Return
  0 = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for a matching filename. The filename in the FCB can contain the ? wild card character to match any character.

If a matching directory entry is found, it is deleted from the directory. If the ? wild card character is used in the filename, all matching directory entries are deleted. AL returns 0.

If no matching directory entry is found, AL returns FFH (255).

Macro Definition:

```
delete  macro   fcb
        mov     dx,offset fcb
        mov     ah,13H
        int     21H
        endm
```

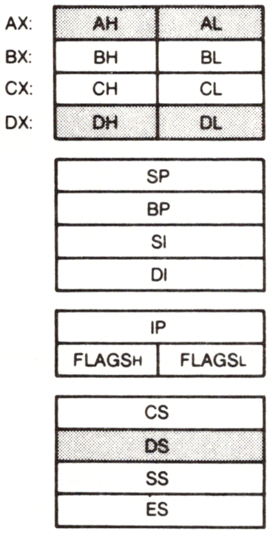
Example

The following program deletes each file on the disk in drive B: that was last written before December 31, 1982:

```
year            dw    1982
month           db    12
day             db    31
files           db    0
ten             db    10
message         db    "NO FILES DELETED.",13,10,"$"
                                  ;see Function 09H for
                                  ;explanation of $
fcb             db    2,"???????????"
                db    25 dup  (?)
```

```
buffer          db    128 dup (?)
                .
                .
func_13H:       set_dta  buffer         ;see Function 1AH
                search_first fcb        ;see Function 11H
                cmp  al,FFH             ;directory entry found?
                je   all_done           ;no, no files on disk
compare:        convert_date buffer     ;see end of chapter
                cmp  cx,year            ;next several lines
                jg   next               ;check date in directory
                cmp  dl,month           ;entry against date
                jg   next               ;above & check next file
                cmp  dh,day             ;if date in directory
                jge  next               ;entry isn't earlier.
                delete  buffer          ;THIS FUNCTION
                inc  files              ;bump deleted-files
                                        ;counter
next:           search_next fcb         ;see Function 12H
                cmp  al,00H             ;directory entry found?
                je   compare            ;yes, check date
                cmp  files,0            ;any files deleted?
                je   all_done           ;no, display NO FILES
                                        ;message.
                convert files,ten,message ;see end of chapter
all_done:       display message         ;see Function 09H
```

Sequential Read (Function 14H)



Call
AH = 14H
DS:DX
  Opened FCB

Return
AL
  0 = Read completed successfully
  1 = EOF
  2 = DTA too small
  3 = EOF, partial record

DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by the current block (offset 0CH) and Current Record (offset 20H) fields is loaded at the Disk Transfer Address, then the Current Block and Current Record fields are incremented.

The record size is set to the value at offset 0EH in the FCB.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file, no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

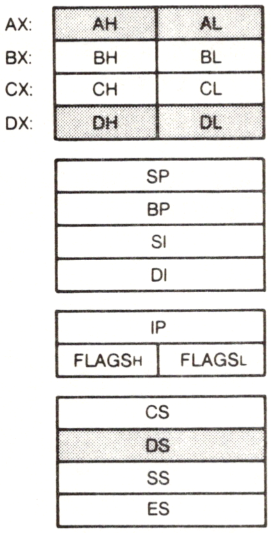
Macro Definition:
```
read_seq macro fcb
         mov   dx,offset fcb
         mov   ah,14H
         int   21H
         endm
```

Example

The following program displays the file named TEXTFILE.ASC that is on the disk in drive B:; its function is similar to the MS-DOS TYPE command. If a partial record is in the buffer at end of file, the routine that displays the partial record displays characters until it encounters an end-of-file mark (ASCII 26, or CONTROL-Z):

```
fcb            db    2,"TEXTFILEASC"
               db    25  dup (?)
buffer         db    128 dup (?),"$"
               .
               .
func_14H:      set_dta  buffer    ;see Function 1AH
               open  fcb          ;see Function 0FH
read_line:     read_seq  fc       ;THIS FUNCTION
               cmp   al,02H       ;end-of-file?
               je    all_done     ;yes
               cmp   al,02H       ;end-of-file with partial
                                  ;record?
               jg    check_more   ;yes
               display  buffer    ;see Function 09H
               jmp   read_line    ;get another record
check_more:    cmp   al,03H       ;partial record in buffer?
               jne   all_done     ;no, go home
               xor   si,si        ;set index to 0
find_eof:      cmp   buffer[si],26 ;is character EOF?
               je    all_done     ;yes, no more to display
               display_char buffer[si]  ;see Function 02H
               inc   si           ;bump index to next
                                  ;character
               jmp   find_eof     ;check next character
all_done:      close fcb          ;see Function 10H
```

Sequential Write (Function 15H)



Call
AH = 15H
DS:DX
  Opened FCB

Return
AL
  00H = Write completed successfully
  01H = Disk full
  02H = DTA too small

DX must contain the offset (from the segment address in DS) of an opened FCB. The record pointed to by Current Block (offset 0CH) and Current Record (offset 20H) fields is written from the Disk Transfer Address, then the current block and current record fields are incremented.

The record size is set to the value at offset 0EH in the FCB. If the Record Size is less than a sector, the data at the Disk Transfer Address is written to a buffer; the buffer is written to disk when it contains a full sector of data, or the file is closed, or a Reset Disk system call (Function 0DH) is issued.

AL returns a code that describes the processing:

Code	Meaning
0	Transfer completed successfully.
1	Disk full; write canceled.
2	Not enough room at the Disk Transfer Address to write one record; write canceled

Macro Definition:

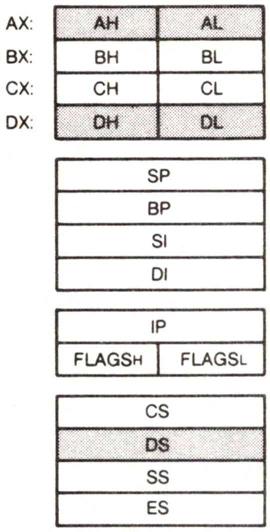
```
write_seq macro fcb
          mov   dx,offset fcb
          mov   ah,15H
          int   21H
          endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B: that contains the disk number (0 = A:, 1 = B:, etc.) and filename from each directory entry on the disk:

```
record_size     equ   14              ;offset of Record Size
                                      ;field in FCB
                .
                .
fcb1            db    2,"DIR     TMP"
                db    25 dup (?)
fcb2            db    2,"???????????"
                db    25 dup (?)
buffer          db    128 dup (?)
                .
                .
func_15H:   set_dta      buffer       ;see Function 1AH
            search_first fcb2         ;see Function 11H
            cmp          al,FFH       ;directory entry found?
            je           all_done     ;no, no files on disk
            create       fcb1         ;see Function 16H
            mov          fcb1[record_size],12
                                      ;set record size to 12
write_it:   write_seq    fcb1         ;THIS FUNCTION
            search_next  fcb2         ;see Function 12H
            cmp          al,FFH       ;directory entry found?
            je           all_done     ;no, go home
            jmp          write_it     ;yes, write the record
all_done:   close        fcb1         ;see Function 10H
```

Create File (Function 16H)



Call
AH = 16H
DS:DX
  Unopened FCB

Return
AL
  00H = Empty directory found
  FFH (255) = No empty directory
        available

DX must contain the offset (from the segment address in DS) of an unopened FCB. The directory is searched for an empty entry or an existing entry for the specified filename.

If an empty directory entry is found, it is initialized to a zero-length file, the Open File system call (Function 0FH) is called, and AL returns 0. You can create a hidden file by using an extended FCB with the attribute byte (offset FCB-1) set to 2.

If an entry is found for the specified filename, all data in the file is released, making a zero-length file, and the Open File system call (Function 0FH) is issued for the filename (in other words, if you try to create a file that already exists, the existing file is erased, and a new, empty file is created).

If an empty directory entry is not found and there is no entry for the specified filename, AL returns FFH (255).

Macro Definition:

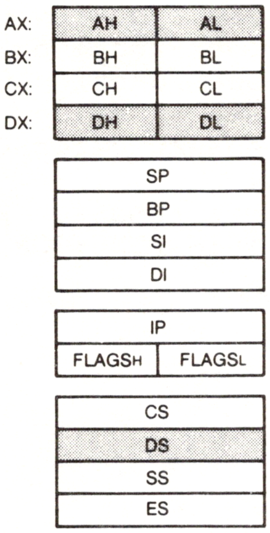
```
create macro fcb
       mov   dx,offset fcb
       mov   ah,16H
       int   21H
       endm
```

Example

The following program creates a file named DIR.TMP on the disk in drive B: that contains the disk number (0 = A:, 1 = B:, etc.) and filename from each directory entry on the disk:

```
record_size    equ   14            ;offset of Record Size
                                   ;field of FCB
               .
               .
fcb1           db    2,"DIR     TMP"
               db    25  dup (?)
fcb2           db    2,"???????????"
               db    25  dup (?)
buffer         db    128 dup (?)
               .
               .
func_16H:      set_dta   buffer     ;see Function 1AH
               search_first  fcb2   ;see Function 11H
               cmp       al,FFH     ;directory entry found?
               je        all_done   ;no, no files on disk
               create    fcb1       ;THIS FUNCTION
               mov       fcb1[record_size],12
                                    ;set record size to 12
write_it:      write_seq fcb1       ;see Function 15H
               search_next  fcb2    ;see Function 12H
               cmp       al,FFH     ;directory entry found?
               je        all_done   ;no, go home
               jmp       write_it   ;yes, write the record
all_done:      close     fcb1       ;see Function 10H
```

Rename File (Function 17H)



Call
AH = 17H
DS:DX
  Modified FCB

Return
AL
  00H = Directory entry found
  FFH (255) = No directory entry
  found or destination already
  exists

DX must contain the offset (from the segment address in DS) of an FCB with the drive number and filename filled in, followed by a second filename at offset 11H. The disk directory is searched for an entry that matches the first filename, which can contain the ? wild card character.

If a matching directory entry is found, the filename in the directory entry is changed to match the second filename in the modified FCB (the two filenames cannot be the same name). If the ? wild card character is used in the second filename, the corresponding characters in the filename of the directory entry are not changed. AL returns 0.

If a matching directory entry is not found or an entry is found for the second filename, AL returns FFH (255).

Macro Definition:

```
rename macro fcb,newname
       mov   dx,offset fcb
       mov   ah,17H
       int   21H
       endm
```

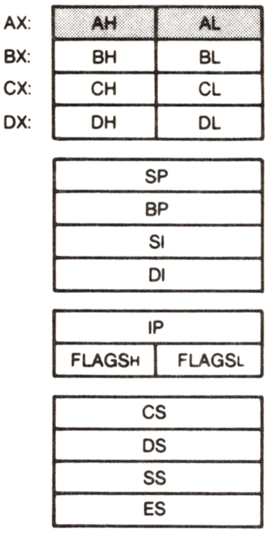
Example

The following program prompts for the name of a file and a new name, then renames the file:

```
fcb                 db    37 dup (?)
prompt1             db   "Filename: $"
prompt2             db   "New name: $"
reply               db    17 dup(?)
crlf                db    13,10,"$"
                    .
                    .
```

```
func_17H:          display  prompt1        ;see Function 09H
                   get_string  15,reply    ;see Function 0AH
                   display  crlf           ;see Function 09H
                   parse    reply[2],fcb   ;see Function 29H
                   display  prompt2        ;see Function 09H
                   get_string  15,reply    ;see Function 0AH
                   display  crlf           ;see Function 09H
                   parse    reply[2],fcb[16]
                                           ;see Function 29H
                   rename   fcb            ;THIS FUNCTION
```

Current Disk (Function 19H)



Call
AH = 19H

Return
AL
Currently selected drive
(0 = A, 1 = B, etc.)

AL returns the currently selected drive (0 = A:, 1 = B:, etc.).

Macro Definition:

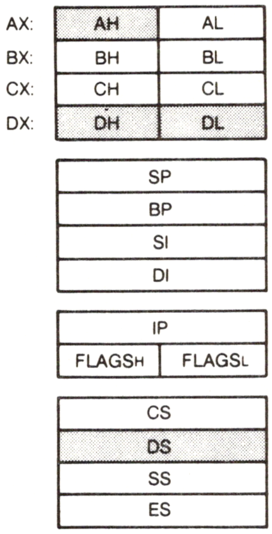
```
current_disk  macro
              mov   ah,19H
              int   21H
              endm
```

Example

The following program displays the currently selected (default) drive in a 2-drive system:

```
message     db  "Current disk is $"   ;see Function 09H
                                      ;for explanation of $
crlf        db   13,10,"$"
            .
            .
func_19H:   display  message          ;see Function 09H
            current_disk              ;THIS FUNCTION
            cmp      al,00H           ;is it disk A?
            jne      disk_b           ;no, it's disk B:
            display_char "A"          ;see Function 02H
            jmp      all_done
disk_b:     display_char "B"          ;see Function 02H
all_done:   display  crlf             ;see Function 09H
```

Set Disk Transfer Address (Function 1AH)



Call
AH = 1AH
DS:DX
 Disk Transfer Address

Return
None

DX must contain the offset (from the segment address in DS) of the Disk Transfer Address. Disk transfers cannot wrap around from the end of the segment to the beginning, nor can they overflow into another segment.

NOTE

If you do not set the Disk Transfer Address, MS-DOS defaults to offset 80H in the Program Segment Prefix.

Macro Definition:

```
set_dta  macro  buffer
         mov    dx,offset buffer
         mov    ah,1AH
         int    21H
         endm
```

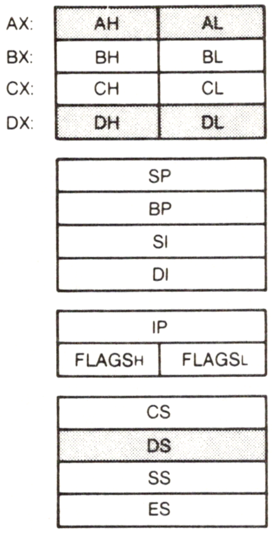
Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B:. The file contains 26 records; each record is 28 bytes long:

```
record_size     equ   14      ;offset of Record Size
                              ;field of FCB
relative_record equ   33      ;offset of Relative Record
                              ;field of FCB
               .
               .
```

```
fcb             db    2,"ALPHABETDAT"
                db    25  dup (?)
buffer          db    34 dup(?),"$"
prompt          db   "Enter letter: $"
crlf            db    13,10,"$"
                .
                .
func_1AH:       set_dta  buffer         ;THIS FUNCTION
                open     fcb            ;see Function 0FH
                mov      fcb[record_size],28 ;set record size
get_char:       display  prompt         ;see Function 09H
                read_kbd_and_echo       ;see Function 01H
                cmp      al,0DH         ;just a CR?
                je       all_done       ;yes, go home
                sub      al,41H         ;convert ASCII
                                        ;code to record
                mov      fcb relative_record ,al
                                        ;set relative record
                display  crlf           ;see Function 09H
                read_ran fcb            ;see Function 21H
                display  buffer         ;see Function 09H
                display  crlf           ;see Function 09H
                jmp      get_char       ;get another character
all_done:       close    fcb            ;see Function 10H
```

Random Read (Function 21H)



Call
AH = 21H
DS:DX
  Opened FCB

Return
AL
  00H = Read completed successfully
  01H = EOF
  02H = DTA too small
  03H = EOF, partial record

DX must contain the offset (from the segment address in DS) of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is loaded at the Disk Transfer Address.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

Macro Definition:

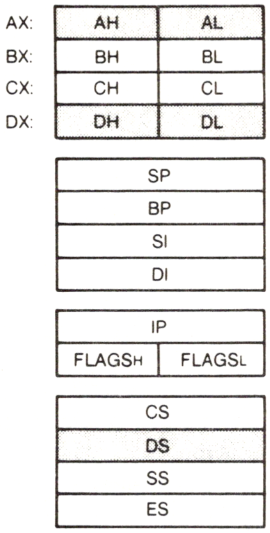
```
read_ran  macro  fcb
          mov    dx,offset fcb
          mov    ah,21H
          int    21H
          endm
```

Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B:. The file contains 26 records; each record is 28 bytes long:

```
record_size     equu   14        ;offset of Record Size
                                 ;field of FCB
relative_record equ    33        ;offset of Relative Record
                                 ;field of FCB
                .
                .
fcb           db    2,"ALPHABETDAT"
              db    25  dup (?)
buffer        db    34 dup(?),"$"
prompt        db   "Enter letter: $"
crlf          db    13,10,"$"
                .
                .
func_21H:     set_dta  buffer              ;see Function 1AH
              open     fcb                 ;see Function 0FH
              mov      fcb[record_size],28 ;set record size
get_char:     display  prompt              ;see Function 09H
              read_kbd_and_echo            ;see Function 01H
              cmp      al,0DH              ;just a CR?
              je       all_done            ;yes, go home
              sub      al,41H              ;convert ASCII code
                                           ;to record #
              mov      fcb[relative_record],al ;set relative
                                           ;record
              display  crlf                ;see Function 09H
              read_ran fcb                 ;THIS FUNCTION
              display  buffer              ;see Function 09H
              display  crlf                ;see Function 09H
              jmp      get_char            ;get another char.
all_done:     close    fcb                 ;see Function 10H
```

Random Write (Function 22H)



Call
AH = 22H
DS:DX
  Opened FCB

Return
AL
  00H = Write completed successfully
  01H = Disk full
  02H = DTA too small

DX must contain the offset from the segment address in DS of an opened FCB. The Current Block (offset 0CH) and Current Record (offset 20H) fields are set to agree with the Relative Record field (offset 21H), then the record addressed by these fields is written from the Disk Transfer Address. If the record size is smaller than a sector (512 bytes), the records are buffered until a sector is ready to write.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully.
1	Disk is full.
2	Not enough room at the Disk Transfer Address to write one record; write canceled.

Macro Definition:

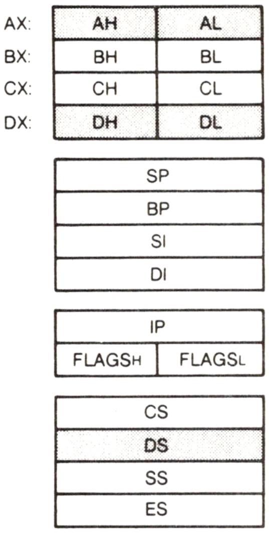
```
write_ran  macro  fcb
           mov    dx,offset fcb
           mov    ah,22H
           int    21H
           endm
```

Example

The following program prompts for a letter, converts the letter to its alphabetic sequence (A = 1, B = 2, etc.), then reads and displays the corresponding record from a file named ALPHABET.DAT on the disk in drive B:. After displaying the record, it prompts the user to enter a changed record. If the user types a new record, it is written to the file; if the user just presses RETURN, the record is not replaced. The file contains 26 records; each record is 28 bytes long:

```
record_size     equ   14        ;offset of Record Size
                                ;field of FCB
relative_record equu  33        ;offset of Relative Record
                                ;field of FCB
           .
           .
fcb        db    2,"ALPHABETDAT"
           db    25  dup (?)
buffer     db    26 dup(?),13,10,"$"
prompt1    db   "Enter letter: $"
prompt2    db   "New record (RETURN for no change): $"
crlf       db    13,10,"$"
reply      db    28 dup (32)
blanks     db    26 dup (32)
           .
           .
func_22H:  set_dta  buffer              ;see Function 1AH
           open     fcb                 ;see Function 0FH
           mov      fcb[record_size],32 ;set record size
get_char:  display  prompt1             ;see Function 09H
           read_kbd_and_echo            ;see Function 01H
           cmp      al,0DH              ;just a CR?
           je       all_done            ;yes, go home
           sub      al,41H              ;convert ASCII
                                        ;code to record #
           mov      fcb[relative_record],al
                                        ;set relative record
           display  crlf                ;see Function 09H
           read_ran fcb                 ;THIS FUNCTION
           display  buffer              ;see Function 09H
           display  crlf                ;see Function 09H
           display  prompt2             ;see Function 09H
           get_string 27,reply          ;see Function 0AH
           display  crlf                ;see Function 09H
           cmp      reply[1],0          ;was anything typed
                                        ;besides CR?
           je       get_char            ;no
                                        ;get another char.
           xor      bx,bx               ;to load a byte
           mov      bl,reply[1]         ;use reply length as
                                        ;counter
           move_string blanks,buffer,26 ;see chapter end
           move_string reply[2],buffer,bx ;see chapter end
           write_ran fcb                ;THIS FUNCTION
           jmp      get_char            ;get another character
all_done:  close    fcb                 ;see Function 10H
```

File Size (Function 23H)



Call
AH = 23H
DS:DX
  Unopened FCB

Return
AL
  00H = Directory entry found
  FFH (255) = No directory entry found

DX must contain the offset (from the segment address in DS) of an unopened FCB. You must set the Record Size field (offset 0EH) to the proper value before calling this function. The disk directory is searched for the first matching entry.

If a matching directory entry is found, the Relative Record field (offset 21H) is set to the number of records in the file, calculated from the total file size in the directory entry (offset 1CH) and the Record Size field of the FCB (offset 0EH). AL returns 00.

If no matching directory is found, AL returns FFH (255).

NOTE

If the value of the Record Size field of the FCB (offset 0EH) doesn't match the actual number of characters in a record, this function does not return the correct file size. If the default record size (128) is not correct, you must set the Record Size field to the correct value before using this function.

Macro Definition:

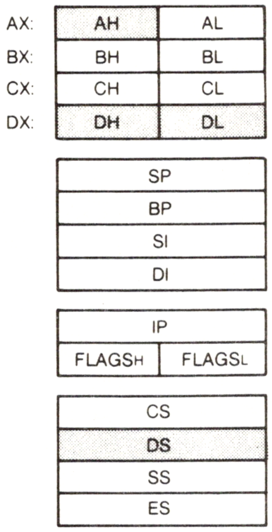
```
file_size macro fcb
          mov   dx,offset fcb
          mov   ah,23H
          int   21H
          endm
```

Example

The following program prompts for the name of a file, opens the file to fill in the Record Size field of the FCB, issues a File Size system call, and displays the file size and number of records in hexadecimal:

```
fcb            db      37 dup  (?)
prompt         db     "File name: $"
msg1           db     "Record length:     ",13,10,"$"
msg2           db     "Records:        ",13,10,"$"
crlf           db      13,10,"$"
reply          db      17 dup(?)
sixteen        db      16
               .
               .
func_23H:      display prompt              ;see Function 09H
               get_string 17,reply         ;see Function 0AH
               cmp     reply[1],0          ;just a CR?
               jne     get_length          ;no, keep going
               jmp     all_done            ;yes, go home
get_length:    display crlf                ;see Function 09H
               parse   reply[2],fcb        ;see Function 29H
               open    fcb                 ;see Function 0FH
               file_size fcb               ;THIS FUNCTION
               mov     si,33               ;offset to Relative
                                           ;Record field
               mov     di,9                ;reply in msg_2
convert_it:    cmp     fcb[si],0           ;digit to convert?
               je      show_it             ;no, prepare message
               convert fcb[si],sixteen,msg_2[di]
               inc     si                  ;bump n-o-r index
               inc     di                  ;bump message index
               jmp     convert_it          ;check for a digit
show_it:       convert fcb[14],sixteen,msg_1[15]
               display msg_1               ;see Function 09H
               display msg_2               ;see Function 09H
               jmp     func_23H            ;get a filename
all_done:      close   fcb                 ;see Function 10H
```

Set Relative Record (Function 24H)



Call
AH = 24H
DS:DX
  Opened FCB

Return
None

DX must contain the offset (from the segment address in DS) of an opened FCB. The Relative Record field (offset 21H) is set to the same file address as the Current Block (offset 0CH) and Current Record (offset 20H) fields.

Macro Definition:

```
set_relative_record  macro   fcb
                     mov     dx,offset fcb
                     mov     ah,24H
                     int     21H
                     endm
```

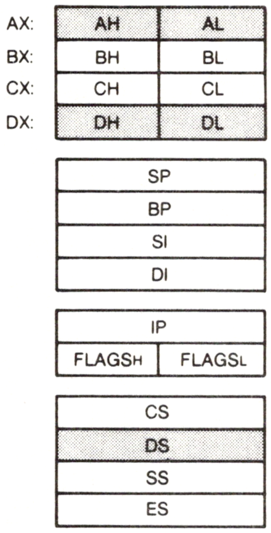
Example

The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by setting the record length equal to the file size and the record count to 1, and using a buffer of 32K bytes. It positions the file pointer by setting the Current Record field (offset 20H) to 1 and using Set Relative Record to make the Relative Record field (offset 21H) point to the same record as the combination of the Current Block (offset 0CH) and Current Record (offset 20H) fields:

```
current_record  equ   32          ;offset of Current Record
                                  ;field of FCB
file_size       equ   16          ;offset of File Size
                                  ;field of FCB
          .
          .
fcb       db      37 dup (?)
filename  db      17 dup(?)
prompt1   db     "File to copy: $"  ;see Function 09H for
prompt2   db     "Name of copy: $"  ;explanation of $
crlf      db      13,10,"$"
```

```
file_length  dw   ?
buffer    db      32767 dup(?)
          .
          .
func_24H: set_dta  buffer                 ;see Function 1AH
          display  prompt1                ;see Function 09H
          get_string  15,filename         ;see Function 0AH
          display  crlf                   ;see Function 09H
          parse    filename[2],fcb        ;see Function 29H
          open     fcb                    ;see Function 0FH
          mov      fcb[current_record],0 ;set Current Record
                                          ;field
          set_relative_record  fcb        ;THIS FUNCTION
          mov      ax,word ptr fcb[file_size] ;get file size
          mov      file_length,ax         ;save it for
                                          ;ran_block_write
          ran_block_read  fcb,1,ax        ;see Function 27H
          display  prompt2                ;see Function 09H
          get_string  15,filename         ;see Function 0AH
          display  crlf                   ;see Function 09H
          parse    filename[2],fcb        ;see Function 29H
          create   fcb                    ;see Function 16H
          mov      fcb[current_record],0 ;set Current Record
                                          ;field
          set_relative_record  fcb        ;THIS FUNCTION
          mov      ax,file_length         ;get original file
                                          ;length
          ran_block_write  fcb,1,ax       ;see Function 28H
          close    fcb                    ;see Function 10H
```

Set Vector (Function 25H)



Call
AH = 25H
AL
  Interrupt number
DS:DX
  Interrupt-handling routine

Return
None

Function 25H should be used to set a particular interrupt vector. The operating system can then manage the interrupts on a per-process basis. Note that programs should never set interrupt vectors by writing them directly in the low memory vector table.

DX must contain the offset (to the segment address in DS) of an interrupt-handling routine. AL must contain the number of the interrupt handled by the routine. The address in the vector table for the specified interrupt is set to DS:DX.

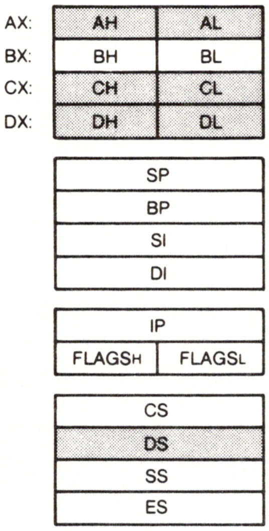
Macro Definition:

```
set_vector  macro  interrupt,seg_addr,off_addr
            mov    al,interrupt
            push   ds
            mov    ax,seg_addr
            mov    ds,ax
            mov    dx,off_addr
            mov    ah,25H
            int    21H
            pop    ds
            endm
```

Example

```
lds     dx,intvector
mov     ah,25H
mov     al,intnumber
int     21H
;There are no errors returned
```

Random Block Read (Function 27H)



Call
AH = 27H
DS:DX
  Opened FCB
CX
  Number of blocks to read

Return
AL
  00H = Read completed successfully
  01H = EOF
  02H = End of segment
  03H = EOF, partial record
CX
  Number of blocks read

DX must contain the offset (to the segment address in DS) of an opened FCB. CX must contain the number of records to read; if it contains 0, the function returns without reading any records (no operation). The specified number of records -- calculated from the Record Size field (offset 0EH) -- is read starting at the record specified by the Relative Record field (offset 21H). The records are placed at the Disk Transfer Address.

AL returns a code that describes the processing:

Code	Meaning
0	Read completed successfully.
1	End-of-file; no data in the record.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.
3	End-of-file; a partial record was read and padded to the record length with zeros.

CX returns the number of records read; the Current Block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

Macro Definition:

```
ran_block_read  macro  fcb,count,rec_size
                mov    dx,offset fcb
                mov    cx,count
                mov    word ptr fcb[14],rec_size
                mov    ah,27H
                int    21H
                endm
```

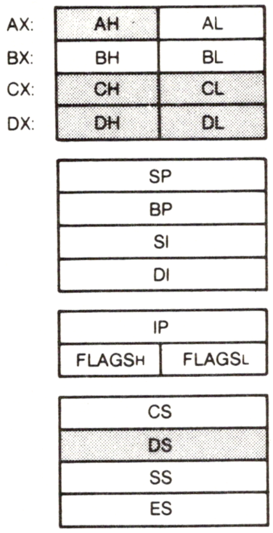
Example

The following program copies a file using the Random Block Read system call. It speeds the copy by specifying a record count of 1 and a record length equal to the file size, and using a buffer of 32K bytes; the file is read as a single record (compare to the sample program for Function 28H that specifies a record length of 1 and a record count equal to the file size):

```
current_record  equ  32   ;offset of Current Record field
file_size       equ  16   ;offset of File Size field
          .
          .
fcb       db      37 dup (?)
filename  db      17 dup(?)
prompt1   db     "File to copy: $"   ;see Function 09H for
prompt2   db     "Name of copy: $"   ;explanation of $
crlf      db      13,10,"$"
file_length  dw   ?
buffer    db      32767 dup(?)
          .
          .
func_27H: set_dta    buffer           ;see Function 1AH
          display    prompt1          ;see Function 09H
          get_string 15,filename      ;see Function 0AH
          display    crlf             ;see Function 09H
          parse      filename[2],fcb  ;see Function 29H
          open       fcb              ;see Function 0FH
          mov        fcb[current_record],0  ;set Current
                                      ;Record field
          set_relative_record fcb     ;see Function 24H
          mov        ax, word ptr fcb[file_size]
                                      ;get file size
          mov        file_length,ax   ;save it for
                                      ;ran_block_write
          ran_block_read  fcb,1,ax    ;THIS FUNCTION
          display    prompt2          ;see Function 09H
          get_string 15,filename      ;see Function 0AH
          display    crlf             ;see Function 09H
          parse      filename[2],fcb  ;see Function 29H
          create     fcb              ;see Function 16H
          mov        fcb[current_record],0
                                      ;set Current Record
                                      ;field
          set_relative_record fcb     ;see Function 24H
```

```
        mov          ax, file_length   ;get original file
                                       ;size
        ran_block_write  fcb,1,ax      ;see Function 28H
        close        fcb               ;see Function 10H
```

Random Block Write (Function 28H)



Call
AH = 28H
DS:DX
  Opened FCB
CX
  Number of blocks to write
  (0 = set File Size field)

Return
AL
  00H = Write completed successfully
  01H = Disk full
  02H = End of segment
CX
  Number of blocks written

DX must contain the offset (to the segment address in DS) of an opened FCB; CX must contain either the number of records to write or 0. The specified number of records (calculated from the Record Size field, offset 0EH) is written from the Disk Transfer Address. The records are written to the file starting at the record specified in the Relative Record field (offset 21H) of the FCB. If CX is 0, no records are written, but the File Size field of the directory entry (offset 1CH) is set to the number of records specified by the Relative Record field of the FCB (offset 21H); allocation units are allocated or released, as required.

AL returns a code that describes the processing:

Code	Meaning
0	Write completed successfully.
1	Disk full. No records written.
2	Not enough room at the Disk Transfer Address to read one record; read canceled.

CX returns the number of records written; the Current Block (offset 0CH), Current Record (offset 20H), and Relative Record (offset 21H) fields are set to address the next record.

Macro Definition:

```
ran_block_write  macro  fcb,count,rec_size
                 mov    dx,offset fcb
                 mov    cx,count
                 mov    word ptr fcb[14],rec_size
                 mov    ah,28H
                 int    21H
                 endm
```

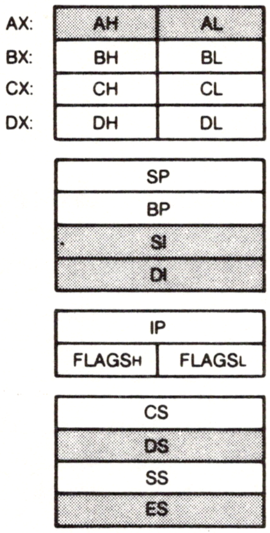
Example

The following program copies a file using the Random Block Read and Random Block Write system calls. It speeds the copy by specifying a record count equal to the file size and a record length of 1, and using a buffer of 32K bytes; the file is copied quickly with one disk access each to read and write (compare to the sample program of Function 27H, that specifies a record count of 1 and a record length equal to file size):

```
current_record  equ  32    ;offset of Current Record field
file_size       equ  16    ;offset of File Size field
          .
          .
fcb       db     37 dup (?)
filename  db     17 dup(?)
prompt1   db    "File to copy: $"   ;see Function 09H for
prompt2   db    "Name of copy: $"   ;explanation of $
crlf      db     13,10,"$"
num_recs  dw     ?
buffer    db     32767 dup(?)
          .
          .
func_28H: set_dta    buffer       ;see Function 1AH
          display    prompt1      ;see Function 09H
          get_string 15,filename  ;see Function 0AH
          display    crlf         ;see Function 09H
          parse      filename[2],fcb  ;see Function 29H
          open       fcb              ;see Function 0FH
          mov        fcb[current_record],0
                                      ;set Current Record
                                      ;field
          set_relative_record fcb     ;see Function 24H
          mov        ax, word ptr fcb[file_size]
                                      ;get file size
          mov        num_recs,ax      ;save it for
                                      ;ran_block_write
          ran_block_read  fcb,num_recs,1  ;THIS FUNCTION
          display    prompt2          ;see Function 09H
          get_string 15,filename      ;see Function 0AH
          display    crlf             ;see Function 09H
          parse      filename[2],fcb  ;see Function 29H
          create     fcb              ;see Function 16H
          mov        fcb[current_record],0  ;set Current
                                      ;Record field
```

```
set_relative_record fcb     ;see Function 24H
mov        ax, file_length  ;get size of original
ran_block_write fcb,num_recs,1 ;see Function 28H
close      fcb              ;see Function 10H
```

Parse File Name (Function 29H)



Call
AH = 29H
AL
  Controls parsing (see text)
DS:SI
  String to parse
ES:DI
  Unopened FCB

Return
AL
  00H = No wild card characters
  01H = Wild-card characters used
  FFH (255) = Drive letter invalid
DS:SI
  First byte past string that was parsed
ES:DI
  Unopened FCB

SI must contain the offset (to the segment address in DS) of a string (command line) to parse; DI must contain the offset (to the segment address in ES) of an unopened FCB. The string is parsed for a filename of the form d:filename.ext; if one is found, a corresponding unopened FCB is created at ES:DI.

Bits 0-3 of AL control the parsing and processing. Bits 4-7 are ignored:

Bit	Value	Meaning
0	0	All parsing stops if a file separator is encountered.
	1	Leading separators are ignored.
1	0	The drive number in the FCB is set to 0 (default drive) if the string does not contain a drive number.
	1	The drive number in the FCB is not changed if the string does not contain a drive number.
2	1	The filename in the FCB is not changed if the string does not contain a filename.
	0	The filename in the FCB is set to 8 blanks if the string does not contain a filename.
3	1	The extension in the FCB is not changed if the string does not contain an extension.
	0	The extension in the FCB is set to 3 blanks if the string does not contain an extension.

If the filename or extension includes an asterisk (*), all remaining characters in the name or extension are set to question mark (?).

Filename separators:

: . ; , = + / "] [\ < > | space tab

Filename terminators include all the filename separators plus any control character. A filename cannot contain a filename terminator; if one is encountered, parsing stops.

If the string contains a valid filename:

1. AL returns 1 if the filename or extension contains a wild card character (* or ?); AL returns 0 if neither the filename nor extension contains a wild card character.

2. DS:SI point to the first character following the string that was parsed.

   ES:DI point to the first byte of the unopened FCB.

If the drive letter is invalid, AL returns FFH (255). If the string does not contain a valid filename, ES:DI+1 points to a blank (ASCII 32).

Macro Definition:

```
parse macro string,fcb
      mov   si,offset string
      mov   di,offset fcb
      push  es
      push  ds
      pop   es
      mov   al,0FH  ;bits 0, 1, 2, 3 on
      mov   ah,29H
      int   21H
      pop   es
      endm
```

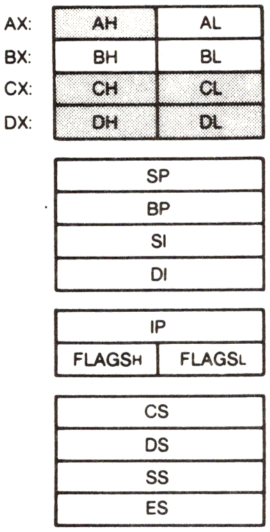
Example

The following program verifies the existence of the file named in reply to the prompt:

```
fcb             db       37 dup (?)
prompt          db      "Filename: $"
reply           db       17 dup(?)
yes             db      "FILE EXISTS",13,10,"$"
no              db      "FILE DOES NOT EXIST",13,10,"$"
                .
                .
```

```
func_29H:      display    prompt          ;see Function 09H
               get_string 15,reply        ;see Function 0AH
               parse      reply[2],fcb    ;THIS FUNCTION
               search_first  fcb          ;see Function 11H
               cmp        al,FFH          ;dir. entry found?
               je         not_there       ;no
               display    yes             ;see Function 09H
               jmp        continue
not_there:     display    no
continue:      .
               .
```

Get Date (Function 2AH)



Call
AH = 2AH

Return
CX
Year (1980 - 2099)
DH
Month (1 - 12)
DL
Day (1 - 31)
AL
Day of week (0=Sun., 6=Sat.)

This function returns the current date set in the operating system as binary numbers in CX and DX:

CX Year (1980-2099)
DH Month (1 = January, 2 = February, etc.)
DL Day (1-31)
AL Day of week (0 = Sunday, 1 = Monday, etc.)

Macro Definition:

```
get_date  macro
          mov    ah,2AH
          int    21H
          endm
```

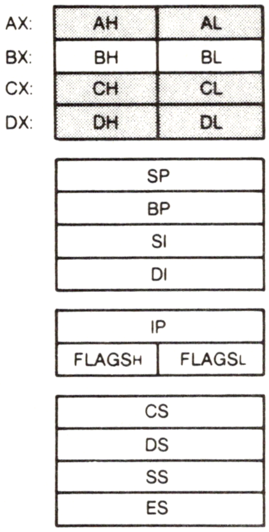
Example

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date:

```
month       db      31,28,31,30,31,30,31,31,30,31,30,31
            .
            .
func_2AH:   get_date                ;see above
            inc     dl              ;increment day
            xor     bx,bx           ;so BL can be used as index
            mov     bl,dh           ;move month to index register
            dec     bx              ;month table starts with 0
            cmp     dl,month[bx]    ;past end of month?
            jle     month_ok        ;no, set the new date
            mov     dl,1            ;yes, set day to 1
            inc     dh              ;and increment month
            cmp     dh,12           ;past end of year?
```

```
           jle    month_ok       ;no, set the new date
           mov    dh,1           ;yes, set the month to 1
           inc    cx             ;increment year
month_ok:  set_date cx,dh,dl    ;THIS FUNCTION
```

Set Date (Function 2BH)



Call
AH = 2BH
CX
  Year (1980 - 2099)
DH
  Month (1 - 12)
DL
  Day (1 - 31)

Return
AL
  00H = Date was valid
  FFH (255) = Date was invalid

Registers CX and DX must contain a valid date in binary:

CX Year (1980-2099)
DH Month (1 = January, 2 = February, etc.)
DL Day (1-31)

If the date is valid, the date is set and AL returns 0. If the date is not valid, the function is canceled and AL returns FFH (255).

Macro Definition:

```
set_date  macro  year,month,day
          mov    cx,year
          mov    dh,month
          mov    dl,day
          mov    ah,2BH
          int    21H
          endm
```

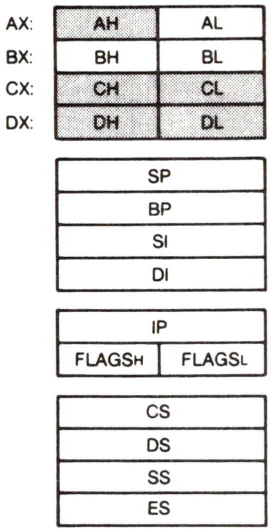
Example

The following program gets the date, increments the day, increments the month or year, if necessary, and sets the new date:

```
month      db      31,28,31,30,31,30,31,31,30,31,30,31
           .
           .
func_2BH:  get_date               ;see Function 2AH
           inc     dl             ;increment day
           xor     bx,bx          ;so BL can be used as index
           mov     bl,dh          ;move month to index register
           dec     bx             ;month table starts with 0
           cmp     dl,month[bx]   ;past end of month?
           jle     month_ok       ;no, set the new date
```

```
             mov     dl,1          ;yes, set day to 1
             inc     dh            ;and increment month
             cmp     dh,12         ;past end of year?
             jle     month_ok      ;no, set the new date
             mov     dh,1          ;yes, set the month to 1
             inc     cx            ;increment year
month_ok:    set_date cx,dh,dl     ;THIS FUNCTION
```

Get Time (Function 2CH)



Call
AH = 2CH

Return
CH
  Hour (0 - 23)
CL
  Minutes (0 - 59)
DH
  Seconds (0 - 59)
DL
  Hundredths (0 - 99)

This function returns the current time set in the operating system as binary numbers in CX and DX:

CH	Hour (0-23)
CL	Minutes (0-59)
DH	Seconds (0-59)
DL	Hundredths of a second (0-99)

Macro Definition:

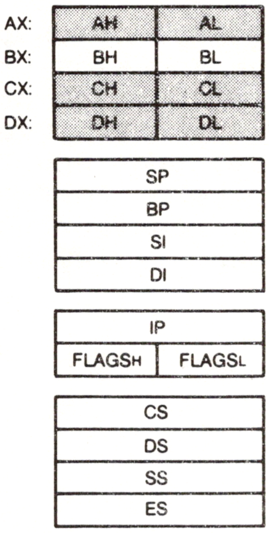
```
get_time  macro
          mov    ah,2CH
          int    21H
          endm
```

Example

The following program continuously displays the time until any key is pressed:

```
time            db     "00:00:00.00",13,10,"$"
ten             db      10
                .
                .
func_2CH:       get_time                  ;THIS FUNCTION
                convert ch,ten,time       ;see end of chapter
                convert cl,ten,time[3]    ;see end of chapter
                convert dh,ten,time[6]    ;see end of chapter
                convert dl,ten,time[9]    ;see end of chapter
                display time              ;see Function 09H
                check_kbd_status          ;see Function 0BH
                cmp      al,FFH           ;has a key been pressed?
                je       all_done         ;yes, terminate
                jmp      func_2CH         ;no, display time
```

Set Time (Function 2DH)



Call

AH = 2DH
CH
  Hour (0 - 23)
CL
  Minutes (0 - 59)
DH
  Seconds (0 - 59)
DL
  Hundredths (0 - 99)

Return

AL
  00H = Time was valid
  FFH (255) = Time was invalid

Registers CX and DX must contain a valid time in binary:

CH Hour (0-23)
CL Minutes (0-59)
DH Seconds (0-59)
DL Hundredths of a second (0-99)

If the time is valid, the time is set and AL returns 0. If the time is not valid, the function is canceled and AL returns FFH (255).

Macro Definition:

```
set_time  macro  hour,minutes,seconds,hundredths
          mov    ch,hour
          mov    cl,minutes
          mov    dh,seconds
          mov    dl,hundredths
          mov    ah,2DH
          int    21H
          endm
```

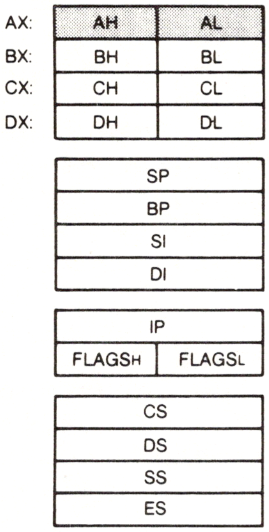
Example

The following program sets the system clock to 0 and continuously displays the time. When a character is typed, the display freezes; when another character is typed, the clock is reset to 0 and the display starts again:

```
time          db  "00:00:00.00",13,10,"$"
ten           db   10
              .
              .
func_2DH:     set_time  0,0,0,0       ;THIS FUNCTION
read_clock:   get_time                ;see Function 2CH
```

```
              convert  ch,ten,time    ;see end of chapter
              convert  cl,ten,time[3] ;see end of chapter
              convert  dh,ten,time[6] ;see end of chapter
              convert  dl,ten,time[9] ;see end of chapter
              display  time           ;see Function 09H
              dir_console_io  FFH     ;see Function 06H
              cmp      al,00H         ;was a char. typed?
              jne      stop           ;yes, stop the timer
              jmp      read_clock     ;no keep timer on
stop:         read_kbd                ;see Function 08H
              jmp      func_2DH       ;keep displaying time
```

Set/Reset Verify Flag (Function 2EH)



Call
AH = 2EH
AL
  00H = Do not verify
  01H = Verify

Return
None

AL must be either 1 (verify after each disk write) or 0 (write without verifying). MS-DOS checks this flag each time it writes to a disk.

The flag is normally off; you may wish to turn it on when writing critical data to disk. Because disk errors are rare and verification slows writing, you will probably want to leave it off at other times.

Macro Definition:

```
verify  macro   switch
        mov     al,switch
        mov     ah,2EH
        int     21H
        endm
```

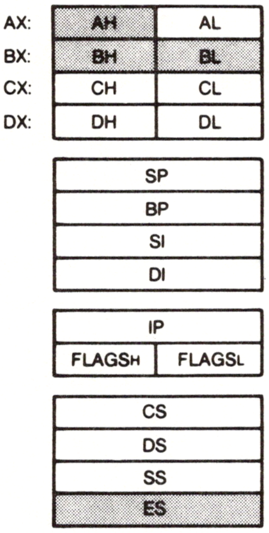
Example

The following program copies the contents of a single-sided disk in drive A: to the disk in drive B:, verifying each write. It uses a buffer of 32K bytes:

```
on              equ    1
off             equ    0
                .
                .
prompt          db     "Source in A, target in B",13,10
                db     "Any key to start. $"
start           dw     0
buffer          db     64 dup (512 dup(?))    ;64 sectors
                .
                .
func_2DH:       display prompt                ;see Function 09H
                read_kbd                      ;see Function 08H
                verify on                     ;THIS FUNCTION
```

```
             mov    cx,5                    ;copy 64 sectors
                                            ;5 times
copy:        push   cx                      ;save counter
             abs_disk_read  0,buffer,64,start
                                            ;see Interrupt 25H
             abs_disk_write  1,buffer,64,start
                                            ;see Interrupt 26H
             add    start,64                ;do next 64 sectors
             pop    cx                      ;restore counter
             loop   copy                    ;do it again
             verify off                     ;THIS FUNCTION

disk_ read  0,buffer,64,start               ;see Interrupt 25H
             abs_disk_write  1,buffer,64,start
                                            ;see Interrupt 26H
             add    start,64                ;do next 64 sectors
             pop    cx                      ;restore counter
             loop   copy                    ;do it again
             verify off
```

Get Disk Transfer Address (Function 2FH)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 2FH

Return
ES:BX
Points to Disk Transfer Address

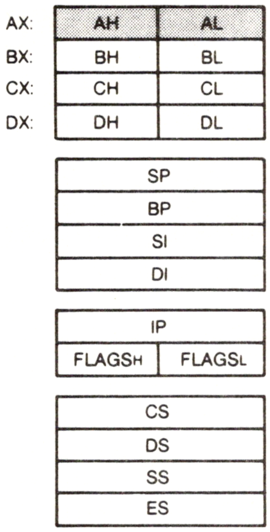
Function 2FH returns the DMA transfer address.

Error returns:
None.

Example

```
        mov     ah,2FH
        int     21H
                ;es:bx has current DMA transfer address
```

Get DOS Version Number (Function 30H)


AX:
BX:
CX:
DX:
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 30H

Return
AL
  Major version number
AH
  Minor version number

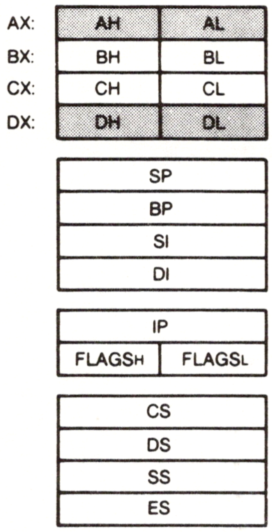
This function returns the MS-DOS version number. On return, AL.AH will be the two-part version designation; i.e., for MS-DOS 1.28, AL would be 1 and AH would be 28. For pre-1.28, DOS AL = 0. Note that version 1.1 is the same as 1.10, not the same as 1.01.

Error returns:
  None.

Example

```
     mov      ah,30H
     int      21H
            ; al is the major version number
            ; ah is the minor version number
            ; bh is the OEM number
            ; bl:cx is the (24 bit) user number
```

Keep Process (Function 31H)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 31H
AL
  Exit code
DX
  Memory size, in paragraphs

Return
None

This call terminates the current process and attempts to set the initial allocation block to a specific size in paragraphs. It will not free up any other allocation blocks belonging to that process. The exit code passed in AX is retrievable by the parent via Function 4DH.

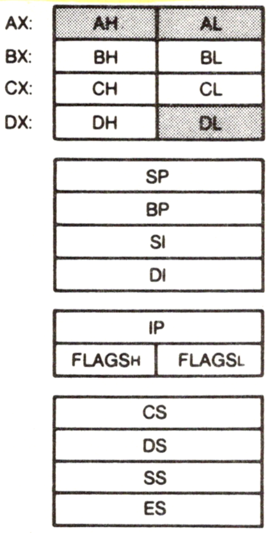
This method is preferred over Interrupt 27H and has the advantage of allowing more than 64K to be kept.

Error returns:
  None.

Example

```
mov     al, exitcode
mov     dx, parasize
mov     ah, 31H
int     21H
```

CONTROL-C Check (Function 33H)



Call

AH = 33H

AL

  Function

  00H = Request current state

  01H = Set state

DL (if setting)

  00H = Off

  01H = On

Return

DL

  00H = Off

  01H = On

MS-DOS ordinarily checks for a CONTROL-C on the controlling device only when doing function call operations 01H-0CH to that device. Function 33H allows the user to expand this checking to include any system call. For example, with the CONTROL-C trapping off, all disk I/O will proceed without interruption; with CONTROL-C trapping on, the CONTROL-C interrupt is given at the system call that initiates the disk operation.

NOTE

> Programs that wish to use calls 06H or 07H to read CONTROL-Cs as data must ensure that the CONTROL-C check is off.

Error return:

AL = FF

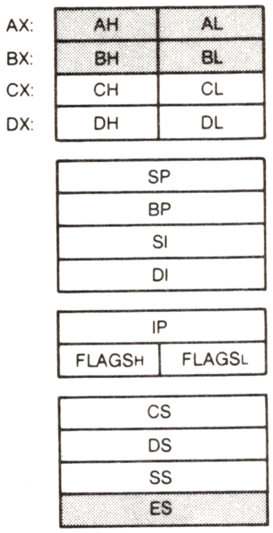
  The function passed in AL was not in the range 0:1.

Example

```
mov     dl,val
mov     ah,33H
mov     al,func
```

```
int     21H
        ; If al was 0, then dl has the current value
        ;of the CONTROL-C check
```

Get Interrupt Vector (Function 35H)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 35H
AL
  Interrupt number

Return
ES:BX
  Pointer to interrupt routine

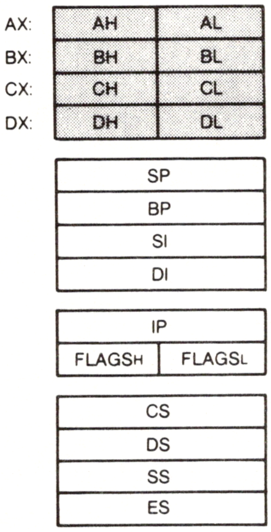
This function returns the interrupt vector associated with an interrupt. Note that programs should never get an interrupt vector by reading the low memory vector table directly.

Error returns:
  None.

Example

```
        mov     ah,35H
        mov     al,interrupt
        int     21H
            ; es:bx now has long pointer to interrupt routine
```

Get Disk Free Space (Function 36H)



Call
AH = 36H
DL
  Drive (0 = Default,
  1 = A, etc.)

Return
BX
  Available clusters
DX
  Clusters per drive
CX
  Bytes per sector
AX
  FFFF if drive number is invalid;
  otherwise sectors per cluster

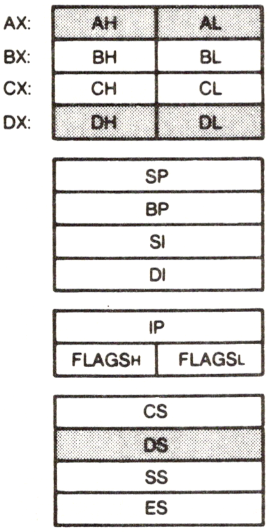
This function returns free space on disk along with additional information about the disk.

Error returns:
AX = FFFF
The drive number given in DL was invalid.

Example

```
mov     ah,36H
mov     dl,Drive              ;0 = default, A = 1
int     21H
    ; bx = Number of free allocation units on drive
    ; dx = Total number of allocation units on drive
    ; cx = Bytes per sector
    ; ax = Sectors per allocation unit
```

Return Country-Dependent Information (Function 38H)



Call
AH = 38H
DS:DX
  Pointer to 32-byte memory area
AL
  Function code. In MS-DOS 2.0, must be 0

Return
Carry set:
AX
  2 = file not found
Carry not set:
  DX:DS filled in with country data

The value passed in AL is either 0 (for current country) or a country code. Country codes are typically the international telephone prefix code for the country.

If DX = -1, then the call sets the current country (as returned by the AL=0 call) to the country code in AL. If the country code is not found, the current country is not changed.

NOTE

> Applications must assume 32 bytes of information. This means the buffer pointed to by DS:DX must be able to accommodate 32 bytes.

This function is fully supported only in versions of MS-DOS 2.01 and higher. It exists in MS-DOS 2.0, but is not fully implemented

This function returns, in the block of memory pointed to by DS:DX, the following information pertinent to international applications:

WORD Date/time format
5 BYTE ASCIZ string currency symbol
2 BYTE ASCIZ string thousands separator
2 BYTE ASCIZ string decimal separator
2 BYTE ASCIZ string date separator
2 BYTE ASCIZ string time separator
1 BYTE Bit field
1 BYTE Currency places
1 BYTE time format
DWORD Case Mapping call
2 BYTE ASCIZ string data list separator

The format of most of these entries is ASCIZ (a NUL terminated ASCII string), but a fixed size is allocated for each field for easy indexing into the table.

The date/time format has the following values:

0 - USA standard h:m:s m/d/y
1 - Europe standard h:m:s d/m/y
2 - Japan standard y/m/d h:m:s

The bit field contains 8 bit values. Any bit not currently defined must be assumed to have a random value.

Bit 0 = 0 If currency symbol precedes the currency amount.
= 1 If currency symbol comes after the currency amount.
Bit 1 = 0 If the currency symbol immediately precedes the currency amount.
= 1 If there is a space between the currency symbol and the amount.

The time format has the following values:

0 - 12 hour time
1 - 24 hour time

The currency places field indicates the number of places which appear after the decimal point on currency amounts.

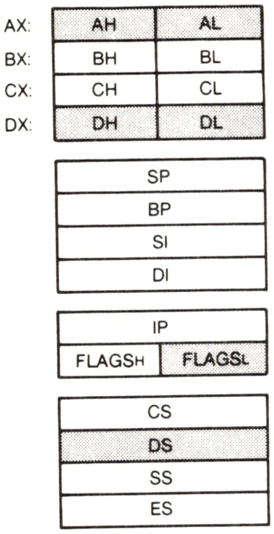
The Case Mapping call is a FAR procedure which will perform country specific lower-to-uppercase mapping on character values from 80H to FFH. It is called with the character to be mapped in AL. It returns the correct upper case code for that character, if any, in AL. AL and the FLAGS are the only registers altered. It is allowable to pass this routine codes below 80H; however nothing is done to characters in this range. In the case where there is no mapping, AL is not altered.

Error returns:
AX
2 = file not found
The country passed in AL was not found (no table for specified country).

Example

```
 lds     dx, blk
 mov     ah, 38H
 mov     al, Country_code
 int     21H
    ;AX = Country code of country returned
```

Create Sub-Directory (Function 39H)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call

AH = 39H
DX:DS
  Pointer to pathname

Return

Carry set:
AX
  3 = path not found
  5 = access denied
Carry not set:
  No error

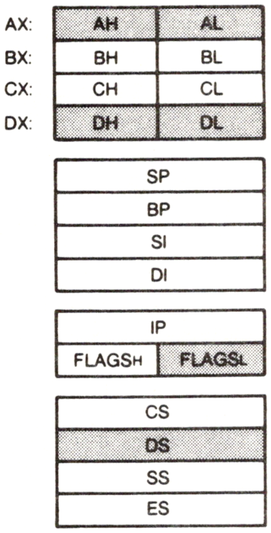
Given a pointer to an ASCIZ name, this function creates a new directory entry at the end.

Error returns:
AX
 3 = path not found
   The path specified was invalid or not found.
 5 = access denied
   The directory could not be created (no room in parent directory), the directory/file already existed or a device name was specified.

Example

```
lds     dx, name
mov     ah, 39H
int     21H
```

Remove a Directory Entry (Function 3AH)



Call
AH = 3AH
DS:DX
  Pointer to pathname

Return
Carry set:
AX
  3 = path not found
  5 = access denied
  16 = current directory
Carry not set:
  No error

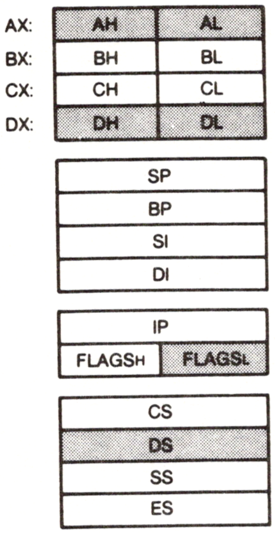
Function 3AH is given an ASCIZ name of a directory. That directory is removed from its parent directory.

Error returns:
AX
3 = path not found
  The path specified was invalid or not found.
5 = access denied
  The path specified was not empty, not a directory, the root directory, or contained invalid information.
16 = current directory
  The path specified was the current directory on a drive.

Example

```
lds     dx, name
mov     ah, 3AH
int     21H
```

Change the Current Directory (Function 3BH)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 3BH
DS:DX
  Pointer to pathname

Return
Carry set:
AX
  3 = path not found
Carry not set:
  No error

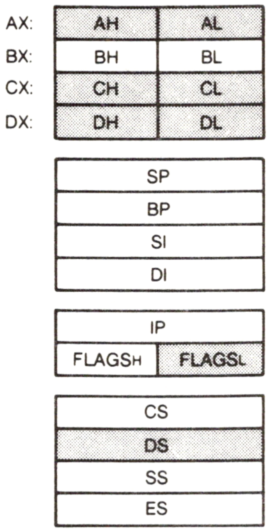
Function 3BH is given the ASCIZ name of the directory which is to become the current directory. If any member of the specified pathname does not exist, then the current directory is unchanged. Otherwise, the current directory is set to the string.

Error returns:
AX
 3 = path not found
   The path specified in DS:DX either indicated a file or the path was invalid.

Example

```
lds     dx, name
mov     ah, 3BH
int     21H
```

Create a File (Function 3CH)



Call
AH = 3CH
DS:DX
  Pointer to pathname
CX
  File attribute

Return
Carry set:
AX
  5 = access denied
  3 = path not found
  4 = too many open files
Carry not set:
  AX is handle number

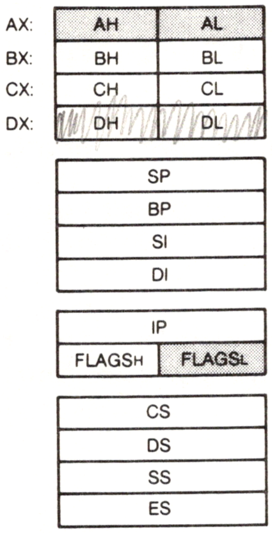
Function 3CH creates a new file or truncates an old file to zero length in preparation for writing. If the file did not exist, then the file is created in the appropriate directory and the file is given the attribute found in CX. The file handle returned has been opened for read/write access.

Error returns:
AX
5 = access denied
  The attributes specified in CX contained one that could not be created (directory, volume ID), a file already existed with a more inclusive set of attributes, or a directory existed with the same name.
3 = path not found
  The path specified was invalid.
4 = too many open files
  The file was created with the specified attributes, but there were no free handles available for the process, or the internal system tables were full.

Example

```
lds     dx, name
mov     ah, 3CH
mov     cx, attribute
int     21H
   ; ax now has the handle
```

Open a File (Function 3DH)



Call
AH = 3DH
AL
  Access
  0 = File opened for reading
  1 = File opened for writing
  2 = File opened for both
  reading and writing
DS:DX POINTER TO PATHNAME

Return
Carry set:
AX
  12 = invalid access
  2 = file not found
  5 = access denied
  4 = too many open files
Carry not set:
  AX is handle number

Function 3DH associates a 16-bit file handle with a file.

The following values are allowed:

ACCESS	Function
0	file is opened for reading
1	file is opened for writing
2	file is opened for both reading and writing.

DS:DX point to an ASCIZ name of the file to be opened.

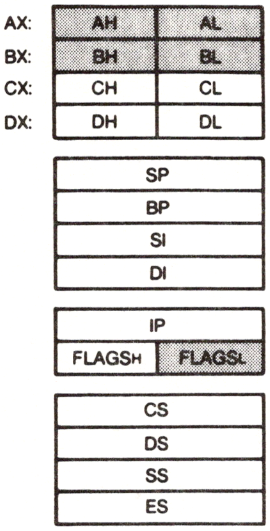
The read/write pointer is set at the first byte of the file and the record size of the file is 1 byte. The returned file handle must be used for subsequent I/O to the file.

Error returns:
AX
12 = invalid access
    The access specified in AL was not in the range 0:2.
2 = file not found
    The path specified was invalid or not found.
5 = access denied
    The user attempted to open a directory or volume-id, or open a read-only file for writing.
4 = too many open files
    There were no free handles available in the current process or the internal system tables were full.

Example

```
lds     dx, name
mov     ah, 3DH
mov     al, access
int     21H
    ; ax has error or file handle
    ; If successful open
```

Close a File Handle (Function 3EH)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 3EH
BX
  File handle

Return
Carry set:
AX
  6 = invalid handle
Carry not set:
  No error

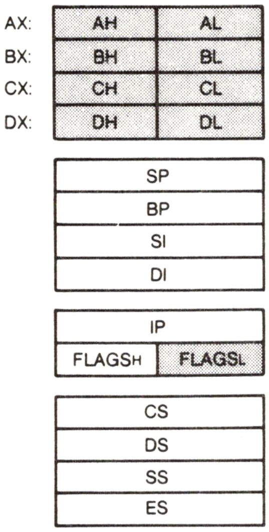
In BX is passed a file handle (like that returned by Functions 3DH, 3CH, or 45H), Function 3EH closes the associated file. Internal buffers are flushed.

Error return:
AX
 6 = invalid handle
  The handle passed in BX was not currently open.

Example

```
        mov     bx, handle
        mov     ah, 3EH
        int     21H
```

Read From File/Device (Function 3FH)



Call
AH = 3FH
DS:DX
  Pointer to buffer
CX
  Bytes to read
BX
  File handle

Return
Carry set:
AX
  Number of bytes read
  6 = invalid handle
  5 = error set:
Carry not set:
  AX = number of bytes read

Function 3FH transfers count bytes from a file into a buffer location. It is not guaranteed that all count bytes will be read; for example, reading from the keyboard will read at most one line of text. If the returned value is zero, then the program has tried to read from the end of file.

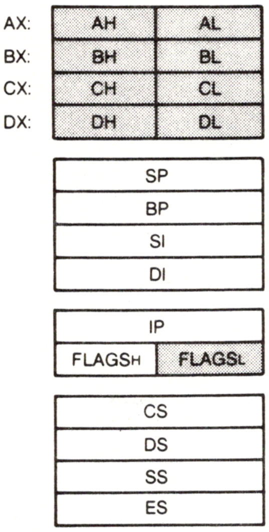
All I/O is done using normalized pointers; no segment wraparound will occur.

Error returns:
AX
 6 = invalid handle
   The handle passed in BX was not currently open.
 5 = access denied
   The handle passed in BX was opened in a mode that did not allow reading.

Example

```
     lds      dx, buf
     mov      cx, count
     mov      bx, handle
     mov      ah, 3FH
     int      21H
        ; ax has number of bytes read
```

Write to a File or Device (Function 40H)



Call
AH = 40H
DS:DX
  Pointer to buffer
CX
  Bytes to write
BX
  File handle

Return
Carry set:
AX
  Number of bytes written
  6 = invalid handle
  5 = access denied
Carry not set:
AX = number of bytes written

Function 40H transfers count bytes from a buffer into a file. It should be regarded as an error if the number of bytes written is not the same as the number requested.

The write system call with a count of zero (CX = 0) will set the file size to the current position. Allocation units are allocated or released as required.

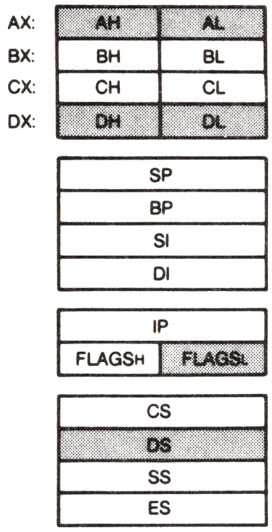
All I/O is done using normalized pointers; no segment wraparound will occur.

Error returns:
AX
 6 = invalid handle
   The handle passed in BX was not currently open.
 5 = access denied
   The handle was not opened in a mode that allowed writing.

Example

```
        lds     dx, buf
        mov     cx, count
        mov     bx, handle
        mov     ah, 40H
        int     21H
           ;ax has number of bytes written
```

Delete a Directory Entry (Function 41H)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 41H
DS:DX
  Pointer to pathname

Return
Carry set:
AX
  2 = file not found
  5 = access denied
Carry not set:
  No error

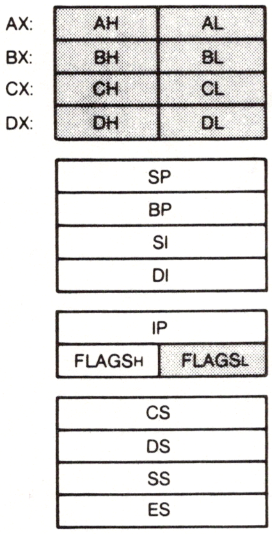
Function 41H removes a directory entry associated with a filename.

Error returns:
AX
 2 = file not found
   The path specified was invalid or not found.
 5 = access denied
   The path specified was a directory or read-only.

Example

```
     lds     dx, name
     mov     ah, 41H
     int     21H
```

Move File Pointer (Function 42H)



Call

AH = 42H
CX:DX
  Distance to move, in bytes
AL
  Method of moving:
  (see text)
BX
  File handle

Return

Carry set:
AX
  6 = invalid handle
  1 = invalid function
Carry not set:
  DX:AX = new pointer location

Function 42H moves the read/write pointer according to one of the following methods:

Method	Function
0	The pointer is moved to offset bytes from the beginning of the file.
1	The pointer is moved to the current location plus offset.
2	The pointer is moved to the end of file plus offset.

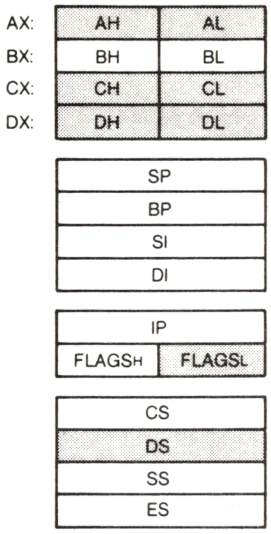
Offset should be regarded as a 32-bit integer with CX occupying the most significant 16 bits.

Error returns:
AX
 6 = invalid handle
  The handle passed in BX was not currently open.
 1 = invalid function
  The function passed in AL was not in the range 0:2.

Example

```
mov     dx, offsetlow
mov     cx, offsethigh
mov     al, method
mov     bx, handle
mov     ah, 42H
int     21H
    ; dx:ax has the new location of the pointer
```

Change Attributes (Function 43H)



Call
AH = 43H
DS:DX
  Pointer to pathname
CX (if AL = 01)
  Attribute to be set
AL
  Function
  01 Set to CX
  00 Return in CX

Return
Carry set:
AX
  3 = path not found
  5 = access denied
  1 = invalid function
Carry not set:
  CX attributes (if AL = 00)

Given an ASCIZ name, Function 42H will set/get the attributes of the file to those given in CX.

A function code is passed in AL:

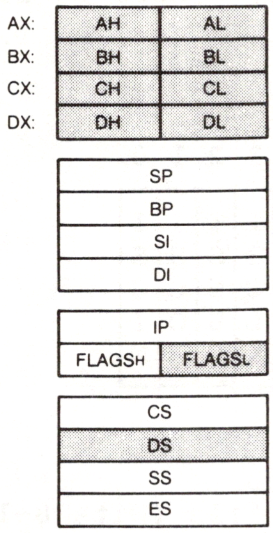
AL	Function
0	Return the attributes of the file in CX.
1	Set the attributes of the file to those in CX.

Error returns:
AX
 3 = path not found
  The path specified was invalid.
 5 = access denied
  The attributes specified in CX contained one that could not be changed (directory, volume ID).
 1 = invalid function
  The function passed in AL was not in the range 0:1.

Example

```
    lds     dx, name
    mov     cx, attribute
    mov     al, func
    int     ah, 43H
    int     21H
```

I/O Control for Devices (Function 44H)



Call
AH = 44H
BX
  Handle
BL
  Drive (for calls AL = 4, 5
  0 = default, 1 = A, etc.)
DS:DX
  Data or buffer
CX
  Bytes to read or write
AL
  Function code; see text

Return
Carry set:
AX
  6 = invalid handle
  1 = invalid function
  13 = invalid data
  5 = access denied
Carry not set:
AL = 2,3,4,5
AX = Count transferred
AL = 6,7
  00 = Not ready
  FF = Ready

Function 44H sets or gets device information associated with an open handle, or sends/receives a control string to a device handle or device.

The following values are allowed for function:

Request	Function
0	Get device information (returned in DX)
1	Set device information (as determined by DX)
2	Read CX number of bytes into DS:DX from device control channel.
3	Write CX number of bytes from DS:DX to device control channel.
4	Same as 2 only drive number in BL 0=default,A:=1,B:=2,...
5	Same as 3 only drive number in BL 0=default,A:=1,B:=2,...
6	Get input status
7	Get output status

This function can be used to get information about device channels. Calls can be made on regular files, but only calls 0,6 and 7 are defined in that case (AL=0,6,7). All other calls return an invalid function error.

Calls AL=0 and AL=1

The bits of DX are defined as follows for calls AL=0 and AL=1. Note that the upper byte MUST be zero on a set call.

15	14	13 12 11 10 9 8	7	6	5	4	3	2	1	0
Res	CTRL	Reserved	ISDEV	EOF	RAW	SPECL	ISCLK	ISNUL	ISCOT	ISCIN

ISDEV = 1 if this channel is a device
= 0 if this channel is a disk file (Bits 8-15 = 0 in this case)

If ISDEV = 1

EOF = 0 if End Of File on input
RAW = 1 if this device is in Raw mode
= 0 if this device is cooked
ISCLK = 1 if this device is the clock device
ISNUL = 1 if this device is the null device
ISCOT = 1 if this device is the console output
ISCIN = 1 if this device is the console input
SPECL = 1 if this device is special

CTRL = 0 if this device can not do control strings via calls AL=2 and AL=3.
CTRL = 1 if this device can process control strings via calls AL=2 and AL=3.
NOTE that this bit cannot be set.

If ISDEV = 0
EOF = 0 if channel has been written
Bits 0-5 are the block device number for the channel (0 = A:, 1 = B:, ...)

Bits 15,8-13,4 are reserved and should not be altered.

Calls 2..5:

These four calls allow arbitrary control strings to be sent or received from a device. The call syntax is the same as the read and write calls, except for 4 and 5, which take a drive number in BL instead of a handle in BX.

An invalid function error is returned if the CTRL bit (see above) is 0.

An access denied is returned by calls AL=4,5 if the drive number is invalid.

Calls 6,7:

These two calls allow the user to check if a file handle is ready for input or output. Status of handles open to a device is the intended use of these calls, but status of a handle open to a disk file is allowed, and is defined as follows:

Input:

Always ready (AL=FF) until EOF reached, then always not ready (AL=0) unless current position changed via LSEEK.

Output:

Always ready (even if disk full).

IMPORTANT

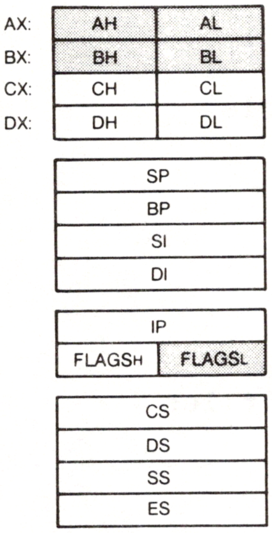
The status is defined at the time the system is CALLED. On future versions, by the time control is returned to the user from the system, the status returned may NOT correctly reflect the true current state of the device or file.

Error returns:
AX
6 = invalid handle
The handle passed in BX was not currently open.
1 = invalid function
The function passed in AL was not in the range 0:7.
13 = invalid data
5 = access denied (calls AL=4..7)

Example

```
     mov      bx, Handle
 (or mov      bl, drive    for calls AL=4,5
                           0=default,A:=1...)
     mov      dx, Data
 (or lds      dx, buf      and
     mov      cx, count    for calls AL=2,3,4,5)
     mov      ah, 44H
     mov      al, func
     int      21H
    ; For calls AL=2,3,4,5 AX is the number of bytes
    ; transferred (same as READ and WRITE).
    ; For calls AL=6,7 AL is status returned, AL=0 if
    ; status is not ready, AL=0FFH otherwise.
```

Duplicate a File Handle (Function 45H)


AX:
BX:
CX:
DX:
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 45H
BX
  File handle

Return
Carry set:
AX
  6 = invalid handle
  4 = too many open files
Carry not set:
AX = new file handle

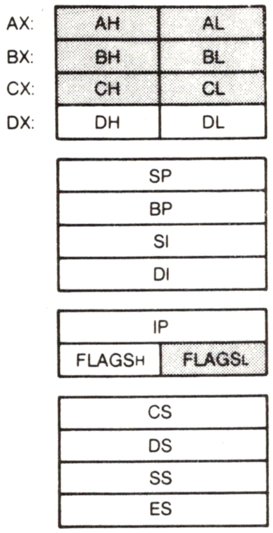
Function 45H takes an already opened file handle and returns a new handle that refers to the same file at the same position.

Error returns:
AX
6 = invalid handle
  The handle passed in BX was not currently open.
4 = too many open files
  There were no free handles available in the current process or the internal system tables were full.

Example

```
    mov     bx, fh
    mov     ah, 45H
    int     21H
       ; ax has the returned handle
```

Force a Duplicate of a Handle (Function 46H)



Call
AH = 46H
BX
  Existing file handle
CX
  New file handle

Return
Carry set:
AX
  6 = invalid handle
  4 = too many open files
Carry not set:
  No error

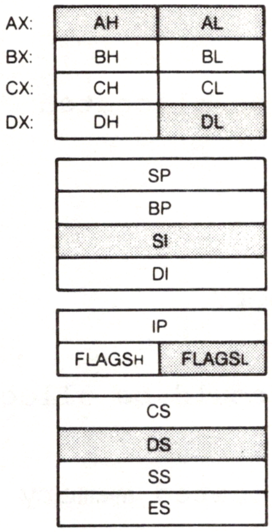
Function 46H takes an already opened file handle and returns a new handle that refers to the same file at the same position. If there was already a file open on handle CX, it is closed first.

Error returns:
AX
 6 = invalid handle
  The handle passed in BX was not currently open.
 4 = too many open files
  There were no free handles available in the current process or the internal system tables were full.

Example

```
    mov     bx, fh
    mov     cx, newfh
    mov     ah, 46H
    int     21H
```

Return Text of Current Directory (Function 47H)



Call
AH = 47H
DS:SI
  Pointer to 64-byte memory area
DL
  Drive number

Return
Carry set:
AX
  15 = invalid drive
Carry not set:
  No error

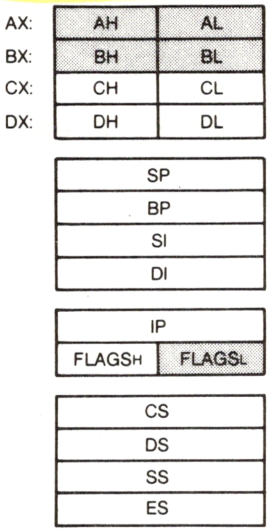
Function 47H returns the current directory for a particular drive. The directory is root-relative and does not contain the drive specifier or leading path separator. The drive code passed in DL is 0=default, 1=A:, 2=B:, etc.

Error returns:
AX
15 = invalid drive
    The drive specified in DL was invalid.

Example

```
        mov     ah, 47H
        lds     si,area
        mov     dl,drive
        int     21H
           ; ds:si is a pointer to 64 byte area that
           ; contains drive current directory.
```

Allocate Memory (Function 48H)



Call
AH = 48H
BX
Size of memory to be allocated

Return
Carry set:
AX
8 = not enough memory
7 = arena trashed
BX
Maximum size that could be allocated
Carry not set:
AX:0
Pointer to the allocated memory

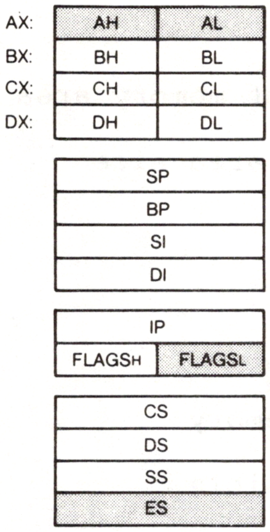
Function 48H returns a pointer to a free block of memory that has the requested size in paragraphs.

Error return:
AX
8 = not enough memory
The largest available free block is smaller than that requested or there is no free block.
7 = arena trashed
The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.

Example

```
mov     bx,size
mov     ah,48H
int     21H
   ; ax:0 is pointer to allocated memory
   ; if alloc fails, bx is the largest block available
```

Free Allocated Memory (Function 49H)



Call
AH = 49H
ES
  Segment address of memory
  area to be freed

Return
Carry set:
AX
  9 = invalid block
  7 = arena trashed
Carry not set:
  No error

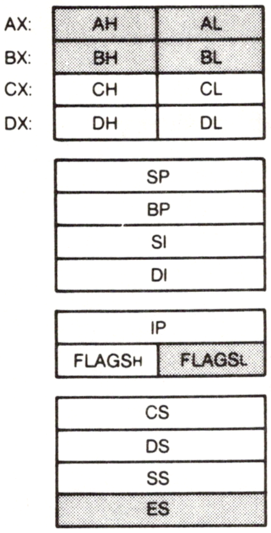
Function 49H returns a piece of memory to the system pool that was allocated by Function Request 49H.

Error return:
AX
9 = invalid block
The block passed in ES is not one allocated via Function Request 49H.
7 = arena trashed
The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.

Example

```
mov     es,block
mov     ah,49H
int     21H
```

Modify Allocated Memory Blocks (Function 4AH)



Call

AH = 4AH
ES
  Segment address of memory area
BX
  Requested memory area size

Return

Carry set:
AX
  9 = invalid block
  7 = arena trashed
  8 = not enough memory
BX
  Maximum size possible
Carry not set:
  No error

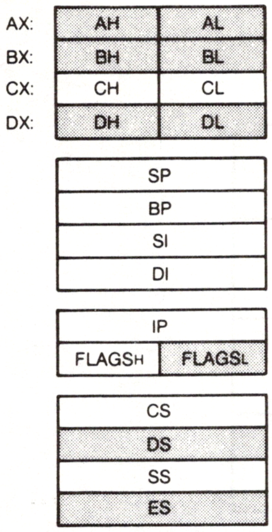
Function 4AH will attempt to grow/shrink an allocated block of memory.

Error return:
AX
9 = invalid block
  The block passed in ES is not one allocated via this function.
7 = arena trashed
  The internal consistency of the memory arena has been destroyed. This is due to a user program changing memory that does not belong to it.
8 = not enough memory
  There was not enough free memory after the specified block to satisfy the grow request.

Example

```
mov     es,block
mov     bx,newsize
mov     ah,4AH
int     21H
    ; if setblock fails for growing, BX will have the
    ; maximum size possible
```

Load and Execute a Program (Function 4BH)



Call
AH = 4BH
DS:DX
  Pointer to pathname
ES:BX
  Pointer to parameter block
AL
  00 = Load and execute program
  03 = Load program

Return
Carry set:
AX
  1 = invalid function
  10 = bad environment
  11 = bad format
  8 = not enough memory
  2 = file not found
Carry not set:
  No error

This function allows a program to load another program into memory and (default) begin execution of it. DS:DX points to the ASCIZ name of the file to be loaded. ES:BX points to a parameter block for the load.

A function code is passed in AL:

AL	Function
0	Load and execute the program. A program header is established for the program and the terminate and CONTROL-C addresses are set to the instruction after the EXEC system call.
3	Load (do not create) the program header, and do not begin execution. This is useful in loading program overlays.

For each value of AL, the block has the following format:

AL = 0 -> load/execute program

WORD segment address of environment.
DWORD pointer to command line at 80H
DWORD pointer to default FCB to be passed at 5CH
DWORD pointer to default FCB to be passed at 6CH

AL = 3 -> load overlay

WORD segment address where file will be loaded.
WORD relocation factor to be applied to the image.

Note that all open files of a process are duplicated in the child process after an EXEC. This is extremely powerful; the parent process has control over the meanings of stdin, stdout, stderr, stdaux and stdprn. The parent could, for example, write a series of records to a file, open the file as standard input, open a listing file as standard output and then EXEC a sort program that takes its input from stdin and writes to stdout.

Also inherited (or passed from the parent) is an "environment." This is a block of text strings (less than 32K bytes total) that convey various configurations parameters. The format of the environment is as follows:

(paragraph boundary)

BYTE ASCIZ string 1
BYTE ASCIZ string 2
...
BYTE ASCIZ string n
BYTE of zero

Typically the environment strings have the form:

parameter=value

For example, COMMAND.COM might pass its execution search path as:

PATH=A:\BIN;B:\BASIC\LIB

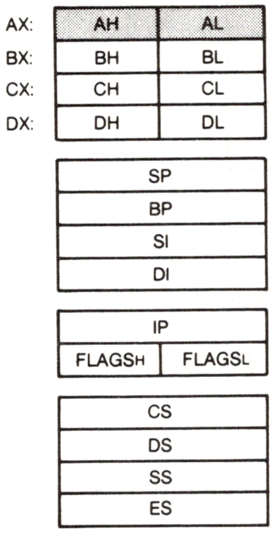
A zero value of the environment address causes the child process to inherit the parent's environment unchanged.

Error returns:
AX
1 = invalid function
  The function passed in AL was not 0, 1 or 3.
10 = bad environment
  The environment was larger than 32Kb.
11 = bad format
  The file pointed to by DS:DX was an EXE format file and contained information that was internally inconsistent.
8 = not enough memory
  There was not enough memory for the process to be created.
2 = file not found
  The path specified was invalid or not found.

Example

```
lds     dx, name
les     bx, blk
mov     ah, 4BH
mov     al, func
int     21H
```

Terminate a Process (Function 4CH)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 4CH
AL
  Return code

Return
None

Function 4CH terminates the current process and transfers control to the invoking process. In addition, a return code may be sent. All files open at the time are closed.

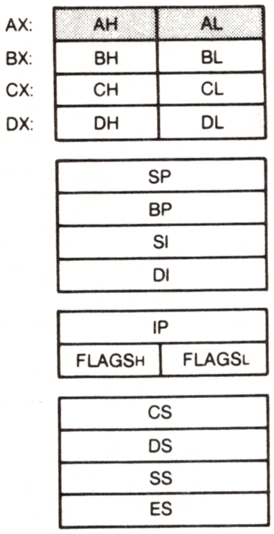
This method is preferred over all others (Interrupt 20H, JMP 0) and has the advantage that CS:0 does not have to point to the Program Header Prefix.

Error returns:
None.

Example

```
mov     al, code
mov     ah, 4CH
int     21H
```

Retrieve the Return Code of a Child (Function 4DH)


AX:
BX:
CX:
DX:
AH
AL
BH
BL
CH
CL
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 4DH

Return
AX
  Exit code

Function 4DH returns the Exit code specified by a child process. It returns this Exit code only once. The low byte of this code is that sent by the Exit routine. The high byte is one of the following:

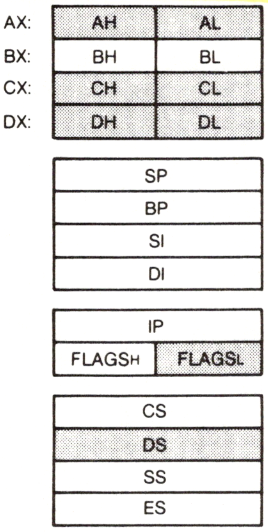
0 - Terminate/abort
1 - CONTROL-C
2 - Hard error
3 - Terminate and stay resident

Error returns:
  None.

Example

```
    mov     ah, 4DH
    int     21H
       ; ax has the exit code
```

Find Match File (Function 4EH)



Call
AH = 4EH
DS:DX
  Pointer to pathname
CX
  Search attributes

Return
Carry set:
AX
  2 = file not found
  18 = no more files
Carry not set:
  No error

Function 4EH takes a pathname with wild-card characters in the last component (passed in DS:DX), a set of attributes (passed in CX) and attempts to find all files that match the pathname and have a subset of the required attributes. A datablock at the current DMA is written that contains information in the following form:

```
find_buf_reserved   DB  21 DUP (?); Reserved*
find_buf_attr       DB  ?  ; attribute found
find_buf_time       DW  ?  ; time
find_buf_date       DW  ?  ; date
find_buf_size_l     DW  ?  ; low(size)
find_buf_size_h     DW  ?  ; high(size)
find_buf_pname      DB  13 DUP (?) ; packed name
find_buf    ENDS
```

*Reserved for MS-DOS use on subsequent find_nexts

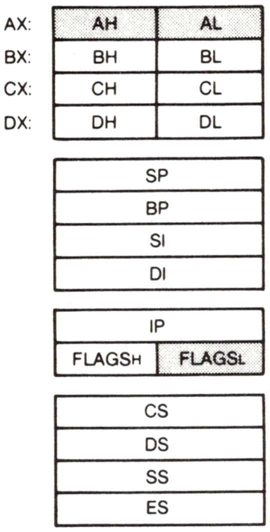
To obtain the subsequent matches of the pathname, see the description of Function 4FH.

Error returns:
AX
 2 = file not found
    The path specified in DS:DX was an invalid path.
18 = no more files
    There were no files matching this specification.

Example

```
mov ah, 4EH
lds dx, pathname
mov cx, attr
int 21H
   ; dma address has datablock
```

Step Through a Directory Matching Files (Function 4FH)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 4FH

Return
Carry set:
AX
  18 = no more files
Carry not set:
  No error

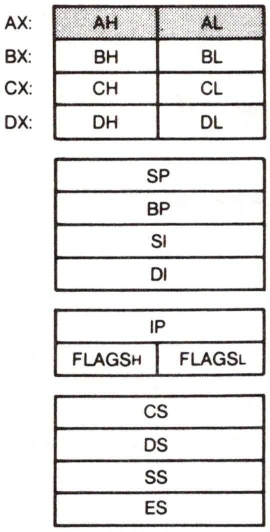
Function 4FH finds the next matching entry in a directory. The current DMA address must point at a block returned by Function 4EH (see Function 4EH).

Error returns:
AX
18 = no more files
        There are no more files matching this pattern.

Example

```
    ; dma points at area returned by Function 4FH
mov ah, 4FH
int 21H
    ; next entry is at dma
```

Return Current Setting of Verify After Write Flag (Function 54H)


AX:
AH
AL
BX:
BH
BL
CX:
CH
CL
DX:
DH
DL
SP
BP
SI
DI
IP
FLAGSH
FLAGSL
CS
DS
SS
ES

Call
AH = 54H

Return
AL
Current verify flag value

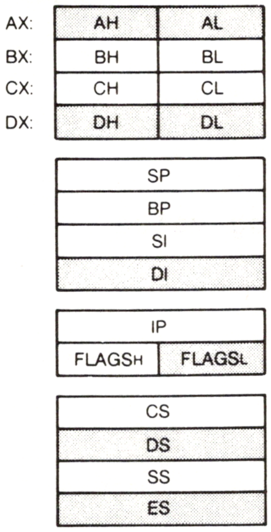
The current value of the verify flag is returned in AL.

Error returns:
None.

Example

```
     mov     ah,54H
     int     21H
         ; al is the current verify flag value
```

Move a Directory Entry (Function 56H)



Call
AH = 56H
DS:DX
  Pointer to pathname of
  existing file
ES:DI
  Pointer to new pathname

Return
Carry set:
AX
  2 = file not found
  17 = not same device
  5 = access denied
Carry not set:
  No error

Function 56H attempts to rename a file into another path. The paths must be on the same device.

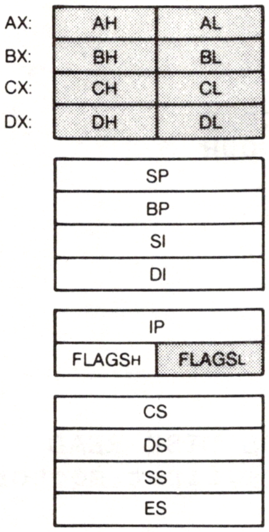
Error returns:
AX
2 = file not found
    The file name specifed by DS:DX was not found.
17 = not same device
    The source and destination are on different drives.
5 = access denied
    The path specified in DS:DX was a directory or the file specified by ES:DI exists or the destination directory entry could not be created.

3 = Path not found

Example

```
    lds     dx, source
    les     di, dest
    mov     ah, 56H
    int     21H
```

Get/Set Date/Time of File (Function 57H)



Call
AH = 57H
AL
  00 = get date and time
  01 = set date and time
BX
  File handle
CX (if AL = 01)
  Time to be set
DX (if AL = 01)
  Date to be set

Return
Carry set:
AX
  1 = invalid function
  6 = invalid handle
Carry not set:
No error
CX/DX set if function 0

Function 57H returns or sets the last-write time for a handle. These times are not recorded until the file is closed.

A function code is passed in AL:

AL	Function
0	Return the time/date of the handle in CX/DX
1	Set the time/date of the handle to CX/DX

Error returns:
AX
 1 = invalid function
   The function passed in AL was not in the range 0:1.
 6 = invalid handle
   The handle passed in BX was not currently open.

Example

```
mov ah, 57H
mov al, func
mov bx, handle
   ; if al = 1 then then next two are mandatory
mov cx, time
mov dx, date
int 21H
   ; if al = 0 then cx/dx has the last write time/date
   ; for the handle.
```

1.8 MACRO DEFINITIONS FOR MS-DOS SYSTEM CALL EXAMPLES

NOTE

These macro definitions apply to system call examples 00H through 57H.

```
;
;*******************
; Interrupts
;*******************
;
                                       ;ABS_DISK_READ
abs_disk_read macro disk,buffer,num_sectors,first_sector
          mov       al,disk
          mov       bx,offset buffer
          mov       cx,num_sectors
          mov       dx,first_sector
          int       37                 ;interrupt 37
          popf
          endm
;
                                       ;ABS_DISK_WRITE
abs_disk_write  macro  disk,buffer,num_sectors,first_sector
          mov       al,disk
          mov       bx,offset buffer
          mov       cx,num_sectors
          mov       dx,first_sector
          int       38                 ;interrupt 38
          popf
          endm
;
stay_resident  macro  last_instruc     ;STAY_RESIDENT
          mov       dx,offset last_instruc
          inc       dx
          int       39                 ;interrupt 39
          endm
;
;*******************
; Functions
;*******************
;
read_kbd_and_echo  macro               ;READ_KBD_AND_ECHO
          mov       ah,1               ;function 1
          int       33
          endm
;
display_char  macro  character         ;DISPLAY_CHAR
          mov       dl,character
          mov       ah,2                ;function 2
          int       33
          endm
;
aux_input macro                         ;AUX_INPUT
          mov       ah,3                ;function 3
          int       33
          endm
;
aux_output macro                        ;AUX_OUTPUT
          mov       ah,4                ;function 4
          int       33
          endm
;;page
print_char  macro   character           ;PRINT_CHAR
          mov       dl,character
          mov       ah,5                ;function 5
          int       33
          endm
;
dir_console_io macro switch             ;DIR_CONSOLE_IO
          mov       dl,switch
          mov       ah,6                ;function 6
          int       33
          endm
;
dir_console_input macro                 ;DIR_CONSOLE_INPUT
          mov       ah,7                ;function 7
          int       33
          endm
;
read_kbd  macro                         ;READ_KBD
          mov       ah,8                ;function 8
          int       33
          endm
;
display   macro     string              ;DISPLAY
          mov       dx,offset string
          mov       ah,9                ;function 9
          int       33
          endm
;
get_string  macro   limit,string        ;GET_STRING
          mov       string,limit
          mov       dx,offset string
          mov       ah,10               ;function 10
          int       33
          endm
;
check_kbd_status macro                  ;CHECK_KBD_STATUS
          mov       ah,11               ;function 11
          int       33
          endm
;
flush_and_read_kbd  macro  switch       ;FLUSH_AND_READ_KBD
```

```
            mov         al,switch
            mov         ah,12                   ;function 12
            int         33
            endm
;
reset_disk macro                                ;RESET DISK
            mov         ah,13                   ;function 13
            int         33
            endm
;;page
select_disk  macro      disk                    ;SELECT_DISK
            mov         dl,disk]-65[
            mov         ah,14                   ;function 14
            int         33
            endm
;
open        macro       fcb                     ;OPEN
            mov         dx,offset fcb
            mov         ah,15                   ;function 15
            int         33
            endm
;
close       macro       fcb                     ;CLOSE
            mov         dx,offset fcb
            mov         ah,16                   ;function 16
            int         33
            endm
;
search_first macro      fcb                     ;SEARCH_FIRST
            mov         dx,offset fcb
            mov         ah,17                   ;Function 17
            int         33
            endm
;
search_next  macro      fcb                     ;SEARCH_NEXT
            mov         dx,offset fcb
            mov         ah,18                   ;function 18
            int         33
            endm
;
delete      macro       fcb                     ;DELETE
            mov         dx,offset fcb
            mov         ah,19                   ;function 19
            int         33
            endm
;
read_seq    macro       fcb                     ;READ_SEQ
            mov         dx,offset fcb
            mov         ah,20                   ;function 20
            int         33
            endm
;
write_seq macro         fcb                     ;WRITE_SEQ
            mov         dx,offset fcb
            mov         ah,21                   ;function 21
```

```
          int       33
          endm
;
create    macro     fcb                 ;CREATE
          mov       dx,offset fcb
          mov       ah,22               ;function 22
          int       33
          endm
;
rename    macro     fcb,newname         ;RENAME
          mov       dx,offset fcb
          mov       ah,23               ;function 23
          int       33
          endm
;
current_disk macro                      ;CURRENT_DISK
          mov       ah,25               ;function 25
          int       33
          endm
;
set_dta   macro     buffer              ;SET_DTA
          mov       dx,offset buffer
          mov       ah,26               ;function 26
          int       33
          endm
;
alloc_table macro                       ;ALLOC_TABLE
          mov       ah,27               ;function 27
          int       33
          endm
;
read_ran  macro     fcb                 ;READ_RAN
          mov       dx,offset fcb
          mov       ah,33               ;function 33
          int       33
          endm
;
write_ran macro     fcb                 ;WRITE_RAN
          mov       dx,offset fcb
          mov       ah,34               ;function 34
          int       33
          endm
;
file_size macro     fcb                 ;FILE_SIZE
          mov       dx,offset fcb
          mov       ah,35               ;function 35
          int       33
          endm
;
set_relative_record  macro  fcb         ;SET_RELATIVE_RECORD
          mov       dx,offset fcb
          mov       ah,36               ;function 36
   int       33
          endm
;;page
```

```
set_vector  macro  interrupt,seg_addr,off_addr  ;SET_VECTOR
          push      ds
          mov       ax,seg_addr
          mov       ds,ax
          mov       dx,off_addr
          mov       al,interrupt
          mov       ah,37              ;function 37
          int       33
          endm
;
create_prog_seg  macro  seg_addr       ;CREATE_PROG_SEG
          mov       dx,seg_addr
          mov       ah,38              ;function 38
          int       33
          endm
;
ran_block_read  macro  fcb,count,rec_size ;RAN_BLOCK_READ
          mov       dx,offset fcb
          mov       cx,count
          mov       word ptr fcb]14[,rec_size
          mov       ah,39              ;function 39
          int       33
          endm
;
ran_block_write  macro  fcb,count,rec_size ;RAN_BLOCK_WRITE
          mov       dx,offset fcb
          mov       cx,count
          mov       word ptr fcb]14[,rec_size
          mov       ah,40                     ;function 40
          int       33
          endm
;
parse     macro     filename,fcb              ;PARSE
          mov       si,offset filename
          mov       di,offset fcb
          push      es
          push      ds
          pop       es
          mov       al,15
          mov       ah,41                     ;function 41
          int       33
          pop       es
          endm
;
get_date  macro                               ;GET_DATE
          mov       ah,42                     ;function 42
          int       33
          endm
;;page
set_date  macro     year,month,day            ;SET_DATE
          mov       cx,year
          mov       dh,month
          mov       dl,day
          mov       ah,43                     ;function 43
          int       33
          endm
;
get_time  macro                                    ;GET_TIME
          mov       ah,44                          ;function 44
          int       33
          endm
;
                                                   ;SET_TIME
set_time  macro     hour,minutes,seconds,hundredths
          mov       ch,hour
          mov       cl,minutes
          mov       dh,seconds
          mov       dl,hundredths
          mov       ah,45                          ;function 45
          int       33
          endm
;
verify    macro     switch                         ;VERIFY
          mov       al,switch
          mov       ah,46                          ;function 46
          int       33
          endm
;
;*******************
; General
;*******************
;
move_string  macro  source,destination,num_bytes
                                          ;MOVE_STRING
          push      es
          mov       ax,ds
          mov       es,ax
          assume    es:data
          mov       si,offset source
          mov       di,offset destination
          mov       cx,num_bytes
      rep movs      es:destination,source
          assume    es:nothing
          pop       es
          endm
;
;
convert   macro     value,base,destination     ;CONVERT
          local     table,start
          jmp       start
table     db       "0123456789ABCDEF"
start:    mov       al,value
          xor       ah,ah
          xor       bx,bx
          div       base
          mov       bl,al
          mov       al,cs:table]bx[
          mov       destination,al
          mov       bl,ah
          mov       al,cs:table]bx[
          mov         destination]1[,al
          endm
;;page
convert_to_binary  macro  string,number,value
                                 ;CONVERT_TO_BINARY
          local       ten,start,calc,mult,no_mult
          jmp         start
ten       db          10
start:    mov         value,0
          xor         cx,cx
          mov         cl,number
          xor         si,si
calc:     xor         ax,ax
          mov         al,string[si]
          sub         al,48
          cmp         cx,2
          jl          no_mult
          push        cx
          dec         cx
mult:     mul         cs:ten
          loop        mult
          pop         cx
no_mult:  add         value,ax
          inc         si
          loop        calc
          endm
;
convert_date macro  dir_entry
          mov         dx,word ptr dir_entry[25]
          mov         cl,5
          shr         dl,cl
          mov         dh,dir_entry[25]
          and         dh,1fh
          xor         cx,cx
          mov         cl,dir_entry[26]
          shr         cl,1
          add         cx,1980
          endm
;
```

1.9 EXTENDED EXAMPLE OF MS-DOS SYSTEM CALLS

```
title DISK DUMP
zero                     equ  0
disk_B                   equ  1
sectors_per_read         equ  9
cr                       equ  13
blank                    equ  32
period                   equ  46
tilde                    equ  126
      INCLUDE  B:CALLS.EQU
;
subttl DATA SEGMENT
page +
data                     segment
;
input_buffer             db  9 dup(512 dup(?))
output_buffer            db  77 dup(" ")
                         db  0DH,0AH,"$"
start_prompt             db  "Start at sector: $"
sectors_prompt           db  "Number of sectors: $"
continue_prompt          db  "RETURN to continue $"
header                   db  "Relative sector $"
end_string               db  0DH,0AH,0AH,07H,"ALL DONE$"
                             ;DELETE THIS
crlf                     db  0DH,0AH,"$"
table                    db  "0123456789ABCDEF$"
;
ten                      db  10
sixteen                  db  16
;
start_sector             dw  1
sector_num            label  byte
sector_number            dw  0
sectors_to_dump          dw  sectors_per_read
sectors_read             dw  0
;
buffer                label  byte
max_length               db  0
current_length           db  0
digits                   db  5 dup(?)
;
data                     ends
;
subttl STACK SEGMENT
page +
stack                    segment   stack
                         dw        100 dup(?)
stack_top                label     word
stack                    ends
;
subttl MACROS
page +
;
```

```
        INCLUDE B:CALLS.MAC
;BLANK LINE
blank_line                macro     number
                          local     print_it
                          push      cx
                          call      clear_line
                          mov       cx,number
print_it:                 display   output_buffer
                          loop      print_it
                          pop       cx
                          endm
;
subttl ADDRESSABILITY
page +
code                      segment
                          assume    cs:code,ds:data,ss:stack
start:                    mov       ax,data
                          mov       ds,ax
                          mov       ax,stack
                          mov       ss,ax
                          mov       sp,offset stack_top
;
                          jmp       main_procedure
subttl PROCEDURES
page +
;
;  PROCEDURES
;  READ_DISK
read_disk                 proc;
                          cmp       sectors_to_dump,zero
                          jle       done
                          mov       bx,offset input_buffer
                          mov       dx,start_sector
                          mov       al,disk_b
                          mov       cx,sectors_per_read
                          cmp       cx,sectors_to_dump
                          jle       get_sector
                          mov       cx,sectors_to_dump
get_sector:               push      cx
                          int       disk_read
                          popf
                          pop       cx
                          sub       sectors_to_dump,cx
                          add       start_sector,cx
                          mov       sectors_read,cx
                          xor       si,si
done:                     ret
read_disk                 endp
;CLEAR_LINE
clear_line                proc;
                          push      cx
                          mov       cx,77
                          xor       bx,bx
move_blank:               mov       output_buffer[bx],' '
                          inc       bx
```

```
                         loop      move_blank
                         pop       cx
                         ret
clear_line               endp
;
;PUT_BLANK
put_blank                proc;
                         mov       output_buffer[di]," "
                         inc       di
                         ret
put_blank                endp
;
;
setup                    proc;
                         display   start_prompt
                         get_string  4,buffer
                         display   crlf
                         convert_to_binary  digits,
                         current_length,start_sector
                         mov       ax,start_sector
                         mov       sector_number,ax
                         display   sectors_prompt
                         get_string  4,buffer
                         convert_to_binary  digits,
                         current_length,sectors_to_dump
                         ret
setup                    endp
;
;CONVERT_LINE
convert_line             proc;
                         push      cx
                         mov       di,9
                         mov       cx,16
convert_it:              convert   input_buffer[si],sixteen,
                         output_buffer[di]
                         inc       si
                         add       di,2
                         call      put_blank
                         loop      convert_it
                         sub       si,16
                         mov       cx,16
                         add       di,4
display_ascii:           mov       output_buffer[di],period
                         cmp       input_buffer[si],blank
                         jl        non_printable
                         cmp       input_buffer[si],tilde
                         jg        non_printable
printable:               mov       dl,input_buffer[si]
                         mov       output_buffer[di],dl
non_printable:           inc       si
                         inc       di
                         loop      display_ascii
                         pop       cx
                         ret
convert_line             endp
```

```
;
;DISPLAY_SCREEN
display_screen            proc;
                          push      cx
                          call      clear_line

;
                          mov       cx,17
;I WANT length header
                          dec       cx
;minus 1 in cx
                          xor       di,di
move_header:              mov       al,header[di]
                          mov       output_buffer[di],al
                          inc       di
                          loop      move_header    ;FIX THIS!
;
                          convert   sector_num[1],sixteen,
                          output_buffer[di]
                          add       di,2
                          convert   sector_num,sixteen,
                          output_buffer[di]
                          display   output_buffer
                          blank_line 2
                          mov       cx,16
dump_it:                  call      clear_line
                          call      convert_line
                          display   output_buffer
                          loop      dump_it
                          blank_line 3
                          display   continue_prompt
                          get_char_no_echo
                          display   crlf
                          pop       cx
                          ret
display_screen            endp
;
;
;   END PROCEDURES
subttl MAIN PROCEDURE
page +
main_procedure:           call      setup
check_done:               cmp       sectors_to_dump,zero
                          jng       all_done
                          call      read_disk
                          mov       cx,sectors_read
display_it:               call      display_screen
                          call      display_screen
                          inc       sector_number
                          loop      display_it
                          jmp       check_done
all_done:                 display   end_string
                          get_char_no_echo
code                      ends
                          end       start
```

CHAPTER 2

MS-DOS 2.0 DEVICE DRIVERS

2.1 WHAT IS A DEVICE DRIVER?

A device driver is a binary file with all of the code in it to manipulate the hardware and provide a consistent interface to MS-DOS. In addition, it has a special header at the beginning that identifies it as a device, defines the strategy and interrupt entry points, and describes various attributes of the device.

NOTE

> For device drivers, the file must not use the ORG 100H (like .COM files). Because it does not use the Program Segment Prefix, the device driver is simply loaded; therefore, the file must have an origin of zero (ORG 0 or no ORG statement).

There are two kinds of device drivers.

1. Character device drivers

2. Block device drivers

Character devices are designed to perform serial character I/O like CON, AUX, and PRN. These devices are named (i.e., CON, AUX, CLOCK, etc.), and users may open channels (handles or FCBs) to do I/O to them.

Block devices are the "disk drives" on the system. They can perform random I/O in pieces called blocks (usually the physical sector size). These devices are not named as the character devices are, and therefore cannot be opened directly. Instead they are identified via the drive letters (A:, B:, C:, etc.).

Block devices also have units. A single driver may be responsible for one or more disk drives. For example, block device driver ALPHA may be responsible for drives A:,B:,C: and D:. This means that it has four units (0-3) defined and, therefore, takes up four drive letters. The position of the driver in the list of all drivers determines which units correspond to which driver letters. If driver ALPHA is the first block driver in the device list, and it defines 4 units (0-3), then they will be A:,B:,C: and D:. If BETA is the second block driver and defines three units (0-2), then they will be E:,F: and G:, and so on. MS-DOS 2.0 is not limited to 16 block device units, as previous versions were. The theoretical limit is 63 (26 - 1), but it should be noted that after 26 the drive letters are unconventional (such as], \, and ^).

NOTE

Character devices cannot define multiple units because they have only one name.

2.2 DEVICE HEADERS

A device header is required at the beginning of a device driver. A device header looks like this:

DWORD pointer to next device (Must be set to -1)
WORD attributes Bit 15 = 1 if char device, 0 is blk. if bit 15 is 1 Bit 0 = 1 if current sti device Bit 1 = 1 if current sto output Bit 2 = 1 if current NUL device Bit 3 = 1 if current CLOCK dev Bit 4 = 1 if special Bits 5-12 Reserved; must be set to 0 Bit 14 is the IOCTL bit Bit 13 is the NON IBM FORMAT bit
WORD pointer to device strategy entry point
WORD pointer to device interrupt entry point
8-BYTE character device name field Character devices set a device name. For block devices the first byte is the number of units

Figure 2. Sample Device Header

Note that the device entry points are words. They must be offsets from the same segment number used to point to this table. For example, if XXX:YYY points to the start of this table, then XXX:strategy and XXX:interrupt are the entry points.

2.2.1 Pointer To Next Device Field

The pointer to the next device header field is a double word field (offset followed by segment) that is set by MS-DOS to point at the next driver in the system list at the time the device driver is loaded. It is important that this field be set to -1 prior to load (when it is on the disk as a file) unless there is more than one device driver in the file. If there is more than one driver in the file, the first word of the double word pointer should be the offset of the next driver's Device Header.

NOTE

> If there is more than one device driver in the .COM file, the last driver in the file must have the pointer to the next Device Header field set to -1.

2.2.2 Attribute Field

The attribute field is used to tell the system whether this device is a block or character device (bit 15). Most other bits are used to give selected character devices certain special treatment. (Note that these bits mean nothing on a block device). For example, assume that a user has a new device driver that he wants to be the standard input and output. Besides installing the driver, he must tell MS-DOS that he wants his new driver to override the current standard input and standard output (the CON device). This is accomplished by setting the attributes to the desired characteristics, so he would set bits 0 and 1 to 1 (note that they are separate!). Similarly, a new CLOCK device could be installed by setting that attribute. (Refer to Section 2.7, "The CLOCK Device," in this chapter for more information.) Although there is a NUL device attribute, the NUL device cannot be reassigned. This attribute exists so that MS-DOS can determine if the NUL device is being used.

The NON IBM FORMAT bit applies only to block devices and affects the operation of the BUILD BPB (Bios Parameter Block) device call. (Refer to Section 2.5.3, "MEDIA CHECK and BUILD BPB," for further information on this call).

The other bit of interest is the IOCTL bit, which has meaning on character and block devices. This bit tells MS-DOS whether the device can handle control strings (via the IOCTL system call, Function 44H).

If a driver cannot process control strings, it should initially set this bit to 0. This tells MS-DOS to return an error if an attempt is made (via Function 44H) to send or receive control strings to this device. A device which can process control strings should initialize the IOCTL bit to 1. For drivers of this type, MS-DOS will make calls to the IOCTL INPUT and OUTPUT device functions to send and receive IOCTL strings.

The IOCTL functions allow data to be sent and received by the device for its own use (for example, to set baud rate, stop bits, and form length), instead of passing data over the device channel as does a normal read or write. The interpretation of the passed information is up to the device, but it must not be treated as a normal I/O request.

2.2.3 Strategy And Interrupt Routines

These two fields are the pointers to the entry points of the strategy and interrupt routines. They are word values, so they must be in the same segment as the Device Header.

2.2.4 Name Field

This is an 8-byte field that contains the name of a character device or the number of units of a block device. If it is a block device, the number of units can be put in the first byte. This is optional, because MS-DOS will fill in this location with the value returned by the driver's INIT code. Refer to Section 2.4, "Installation of Device Drivers" in this chapter for more information.

2.3 HOW TO CREATE A DEVICE DRIVER

In order to create a device driver that MS-DOS can install, you must write a binary file with a Device Header at the beginning of the file. Note that for device drivers, the code should not be originated at 100H, but rather at 0. The link field (pointer to next Device Header) should be -1, unless there is more than one device driver in the file. The attribute field and entry points must be set correctly.

If it is a character device, the name field should be filled in with the name of that character device. The name can be any legal 8-character filename.

MS-DOS always processes installable device drivers before handling the default devices, so to install a new CON device, simply name the device CON. Remember to set the standard input device and standard output device bits in the attribute word on a new CON device. The scan of the device list stops on the first match, so the installable device driver takes precedence.

NOTE

> Because MS-DOS can install the driver anywhere in memory, care must be taken in any far memory references. You should not expect that your driver will always be loaded in the same place every time.

2.4 INSTALLATION OF DEVICE DRIVERS

MS-DOS 2.0 allows new device drivers to be installed dynamically at boot time. This is accomplished by INIT code in the BIOS, which reads and processes the CONFIG.SYS file.

MS-DOS calls upon the device drivers to perform their function in the following manner:

> MS-DOS makes a far call to strategy entry, and passes (in a Request Header) the information describing the functions of the device driver.

This structure allows you to program an interrupt-driven device driver. For example, you may want to perform local buffering in a printer.

2.5 REQUEST HEADER

When MS-DOS calls a device driver to perform a function, it passes a Request Header in ES:BX to the strategy entry point. This is a fixed length header, followed by data pertinent to the operation being performed. Note that it is the device driver's responsibility to preserve the machine state (for example, save all registers on entry and restore them on exit). There is enough room on the stack when strategy or interrupt is called to do about 20 pushes. If more stack is needed, the driver should set up its own stack.

The following figure illustrates a Request Header.

REQUEST HEADER ->

BYTE length of record Length in bytes of this Request Header
BYTE unit code The subunit the operation is for (minor device) (no meaning on character devices)
BYTE command code
WORD status
8 bytes RESERVED

Figure 3. Request Header

2.5.1 Unit Code

The unit code field identifies which unit in your device driver the request is for. For example, if your device driver has 3 units defined, then the possible values of the unit code field would be 0, 1, and 2.

2.5.2 Command Code Field

The command code field in the Request header can have the following values:

Command Code	Function
0	INIT
1	MEDIA CHECK (Block only, NOP for character)
2	BUILD BPB " " " " "
3	IOCTL INPUT (Only called if device has IOCTL)
4	INPUT (read)
5	NON-DESTRUCTIVE INPUT NO WAIT (Char devs only)
6	INPUT STATUS " " "
7	INPUT FLUSH " " "
8	OUTPUT (write)
9	OUTPUT (Write) with verify
10	OUTPUT STATUS " " "
11	OUTPUT FLUSH " " "
12	IOCTL OUTPUT (Only called if device has IOCTL)

2.5.3 MEDIA CHECK And BUILD BPB

MEDIA CHECK and BUILD BPB are used with block devices only.

MS-DOS calls MEDIA CHECK first for a drive unit. MS-DOS passes its current media descriptor byte (refer to the section "Media Descriptor Byte" later in this chapter). MEDIA CHECK returns one of the following results:

> Media Not Changed - current DPB and media byte are OK.
>
> Media Changed - Current DPB and media are wrong. MS-DOS invalidates any buffers for this unit and calls the device driver to build the BPB with media byte and buffer.
>
> Not Sure - If there are dirty buffers (buffers with changed data, not yet written to disk) for this unit, MS-DOS assumes the DPB and media byte are OK (media not changed). If nothing is dirty, MS-DOS assumes the media has changed. It invalidates any buffers for the unit, and calls the device driver to build the BPB with media byte and buffer.
>
> Error - If an error occurs, MS-DOS sets the error code accordingly.

MS-DOS will call BUILD BPB under the following conditions:

> If Media Changed is returned
>
> If Not Sure is returned, and there are no dirty buffers

The BUILD BPB call also gets a pointer to a one-sector buffer. What this buffer contains is determined by the NON IBM FORMAT bit in the attribute field. If the bit is zero (device is IBM format-compatible), then the buffer contains the first sector of the first FAT. The FAT ID byte is the first byte of this buffer. NOTE: The BPB must be the same, as far as location of the FAT is concerned, for all possible media because this first FAT sector must be read before the actual BPB is returned. If the NON IBM FORMAT bit is set, then the pointer points to one sector of scratch space (which may be used for anything).

2.5.4 Status Word

The following figure illustrates the status word in the Request Header.

15	14 13 12 11 10	9	8	7 6 5 4 3 2 1 0
ERR	RESERVED	BUS	DON	ERROR CODE (bit 15 on)

The status word is zero on entry and is set by the driver interrupt routine on return.

Bit 8 is the done bit. When set, it means the operation is complete. For MS-DOS 2.0, the driver sets it to 1 when it exits.

Driver has completed its task.

Bit 15 is the error bit. If it is set, then the low 8 bits indicate the error. The errors are:

0 Write protect violation
1 Unknown Unit
2 Drive not ready
3 Unknown command
4 CRC error
5 Bad drive request structure length
6 Seek error
7 Unknown media
8 Sector not found
9 Printer out of paper
A Write fault
B Read Fault
C General failure

Bit 9 is the busy bit, which is set only by status calls.

> For output on character devices: If bit 9 is 1 on return, a write request (if made) would wait for completion of a current request. If it is 0, there is no current request, and a write request (if made) would start immediately.

> For input on character devices with a buffer: If bit 9 is 1 on return, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the device buffer and a read would return quickly. It also indicates that something has been typed. MS-DOS assumes all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy=0 so that MS-DOS will not continuously wait for something to get into a buffer that does not exist.

One of the functions defined for each device is INIT. This routine is called only once when the device is installed. The INIT routine returns a location (DS:DX), which is a pointer to the first free byte of memory after the device driver (similar to "Keep Process"). This pointer method can be used to delete initialization code that is only needed once, saving on space.

Block devices are installed the same way and also return a first free byte pointer as described above. Additional information is also returned:

> The number of units is returned. This determines logical device names. If the current maximum logical device letter is F at the time of the install call, and the INIT routine returns 4 as the number of units, then they will have logical names G, H, I and J. This mapping is determined by the position of the driver in the device list, and by the number of units on the device (stored in the first byte of the device name field).
>
> A pointer to a BPB (BIOS Parameter Block) pointer array is also returned. There is one table for each unit defined. These blocks will be used to build an internal DOS data structure for each of the units. The pointer passed to the DOS from the driver points to an array of n word pointers to BPBs, where n is the number of units defined. In this way, if all units are the same, all of the pointers can point to the same BPB, saving space. Note that this array must be protected (below the free pointer set by the return) since an internal DOS structure will be built starting at the byte pointed to by the free pointer. The sector size defined must be less than or equal to the maximum sector size defined at default BIOS INIT time. If it isn't, the install will fail.
>
> The last thing that INIT of a block device must pass back is the media descriptor byte. This byte means nothing to MS-DOS, but is passed to devices

so that they know what parameters MS-DOS is currently using for a particular drive unit.

Block devices may take several approaches; they may be dumb or smart. A dumb device defines a unit (and therefore an internal DOS structure) for each possible media drive combination. For example, unit 0 = drive 0 single side, unit 1 = drive 0 double side. For this approach, media descriptor bytes do not mean anything. A smart device allows multiple media per unit. In this case, the BPB table returned at INIT must define space large enough to accommodate the largest possible media supported. Smart drivers will use the media descriptor byte to pass information about what media is currently in a unit.

2.6 FUNCTION CALL PARAMETERS

All strategy routines are called with ES:BX pointing to the Request Header. The interrupt routines get the pointers to the Request Header from the queue that the strategy routines store them in. The command code in the Request Header tells the driver which function to perform.

NOTE

All DWORD pointers are stored offset first, then segment.

2.6.1 INIT

Command code = 0

INIT - ES:BX ->

13-BYTE Request Header
BYTE # of units
DWORD break address
DWORD pointer to BPB array (Not set by character devices)

The number of units, break address, and BPB pointer are set by the driver. On entry, the DWORD that is to be set to the BPB array (on block devices) points to the character after the '=' on the line in CONFIG.SYS that loaded this device. This allows drivers to scan the CONFIG.SYS invocation line for arguments.

NOTE

> If there are multiple device drivers in a single .COM file, the ending address returned by the last INIT called will be the one MS-DOS uses. It is recommended that all of the device drivers in a single .COM file return the same ending address.

2.6.2 MEDIA CHECK

Command Code = 1

MEDIA CHECK - ES:BX ->

13-BYTE Request Header
BYTE media descriptor from DPB
BYTE returned

In addition to setting the status word, the driver must set the return byte to one of the following:

-1 Media has been changed
 0 Don't know if media has been changed
 1 Media has not been changed

If the driver can return -1 or 1 (by having a door-lock or other interlock mechanism) MS-DOS performance is enhanced because MS-DOS does not need to reread the FAT for each directory access.

2.6.3 BUILD BPB (BIOS Parameter Block)

Command code = 2

BUILD BPB - ES:BX ->

13-BYTE Request Header
BYTE media descriptor from DPB
DWORD transfer address (Points to one sector worth of scratch space or first sector of FAT depending on the value of the NON IBM FORMAT bit)
DWORD pointer to BPB

If the NON IBM FORMAT bit of the device is set, then the DWORD transfer address points to a one sector buffer, which can be used for any purpose. If the NON IBM FORMAT bit is 0, then this buffer contains the first sector of the first FAT and the driver must not alter this buffer.

If IBM compatible format is used (NON IBM FORMAT BIT = 0), then the first sector of the first FAT must be located at the same sector on all possible media. This is because the FAT sector will be read BEFORE the media is actually determined. Use this mode if all you want is to read the FAT ID byte.

In addition to setting status word, the driver must set the Pointer to the BPB on return.

In order to allow for many different OEMs to read each other's disks, the following standard is suggested: The information relating to the BPB for a particular piece of media is kept in the boot sector for the media. In particular, the format of the boot sector is:

	3 BYTE near JUMP to boot code	1,2,3
	8 BYTES OEM name and version	4-11
BPB ↓	WORD bytes per sector	12,13
	BYTE sectors per allocation unit	14
	WORD reserved sectors	15,16
	BYTE number of FATs	17
	WORD number of root dir entries	18,19
↑ ?	WORD number of sectors in logical image ?	20,21
BPB ?	BYTE media descriptor	22
	WORD number of FAT sectors	23,24
	WORD sectors per track	25,26
	WORD number of heads	27,28
	WORD number of hidden sectors	29,30

The three words at the end (sectors per track, number of heads, and number of hidden sectors) are optional. They are intended to help the BIOS understand the media. Sectors per track may be redundant (could be calculated from total size of the disk). Number of heads is useful for supporting different multi-head drives which have the same storage capacity, but different numbers of surfaces. Number of hidden sectors may be used to support drive-partitioning schemes.

2.6.4 Media Descriptor Byte

The last two digits of the FAT ID byte are called the media descriptor byte. Currently, the media descriptor byte has been defined for a few media types, including 5-1/4" and 8" standard disks. For more information, refer to Section 3.6, "MS-DOS Standard Disk Formats."

Although these media bytes map directly to FAT ID bytes (which are constrained to the 8 values F8-FF), media bytes can, in general, be any value in the range 0-FF.

2.6.5 READ Or WRITE

Command codes = 3,4,8,9, and 12

READ or WRITE - ES:BX (Including IOCTL) ->

13-BYTE Request Header
BYTE media descriptor from DPB
DWORD transfer address
WORD byte/sector count
WORD starting sector number (Ignored on character devices)

In addition to setting the status word, the driver must set the sector count to the actual number of sectors (or bytes) transferred. No error check is performed on an IOCTL I/O call. The driver must correctly set the return sector (byte) count to the actual number of bytes transferred.

THE FOLLOWING APPLIES TO BLOCK DEVICE DRIVERS:

Under certain circumstances the BIOS may be asked to perform a write operation of 64K bytes, which seems to be a "wrap around" of the transfer address in the BIOS I/O packet. This request arises due to an optimization added to the write code in MS-DOS. It will only manifest on user writes that are within a sector size of 64K bytes on files "growing" past the current EOF. It is allowable for the BIOS to ignore the balance of the write that "wraps around" if it so chooses. For example, a write of 10000H bytes worth of sectors with a transfer address of XXX:1 could ignore the last two bytes. A user program can never request an I/O of more than FFFFH bytes and cannot wrap around (even to 0) in the transfer segment. Therefore, in this case, the last two bytes can be ignored.

2.6.6 NON DESTRUCTIVE READ NO WAIT

Command code = 5

NON DESRUCTIVE READ NO WAIT - ES:BX ->

13-BYTE Request Header
BYTE read from device

If the character device returns busy bit = 0 (characters in buffer), then the next character that would be read is returned. This character is not removed from the input buffer (hence the term "Non Destructive Read"). Basically, this call allows MS-DOS to look ahead one input character.

2.6.7 STATUS

Command codes = 6 and 10

STATUS Calls - ES:BX ->

13-BYTE Request Header

All the driver must do is set the status word and the busy bit as follows:

For output on character devices: If bit 9 is 1 on return, a write request (if made) would wait for completion of a current request. If it is 0, there is no current request and a write request (if made) would start immediately.

For input on character devices with a buffer: A return of 1 means, a read request (if made) would go to the physical device. If it is 0 on return, then there are characters in the devices buffer and a read would return quickly. A return of 0 also indicates that the user has typed something. MS-DOS assumes that all character devices have an input type-ahead buffer. Devices that do not have a type-ahead buffer should always return busy = 0 so that the DOS will not hang waiting for something to get into a buffer which doesn't exist.

2.6.8 FLUSH

Command codes = 7 and 11

FLUSH Calls - ES:BX ->

13-BYTE Request Header

The FLUSH call tells the driver to flush (terminate) all pending requests. This call is used to flush the input queue on character devices.

2.7 THE CLOCK DEVICE

One of the most popular add-on boards is the real time clock board. To allow this board to be integrated into the system for TIME and DATE, there is a special device (determined by the attribute word) called the CLOCK device. The CLOCK device defines and performs functions like any other character device. Most functions will be: "set done bit, reset error bit, return." When a read or write to this device occurs, exactly 6 bytes are transferred. The first two bytes are a word, which is the count of days since 1-1-80. The third byte is minutes; the fourth, hours; the fifth, hundredths of seconds; and the sixth, seconds. Reading the CLOCK device gets the date and time; writing to it sets the date and time.

2.8 EXAMPLE OF DEVICE DRIVERS

The following examples illustrate a block device driver and a character device driver program.

2.8.1 Block Device Driver

```
;********************* A BLOCK DEVICE *******************

        TITLE   5 1/4" DISK DRIVER FOR SCP DISK-MASTER

;This driver is intended to drive up to four 5 1/4" drives
;hooked to the Seattle Computer Products DISK MASTER disk
;controller. All standard IBM PC formats are supported.


FALSE   EQU     0
TRUE    EQU     NOT FALSE

;The I/O port address of the DISK MASTER
DISK    EQU     0E0H
;DISK+0
;       1793    Command/Status
;DISK+1
;       1793    Track
;DISK+2
;       1793    Sector
;DISK+3
;       1793    Data
;DISK+4
;       Aux Command/Status
;DISK+5
;       Wait Sync

;Back side select bit
BACKBIT EQU     04H
;5 1/4" select bit
SMALBIT EQU     10H
;Double Density bit
DDBIT   EQU     08H

;Done bit in status register
DONEBIT EQU     01H

;Use table below to select head step speed.
;Step times for 5" drives
;are double that shown in the table.
;
;Step value    1771    1793
;
;    0          6ms     3ms
;    1          6ms     6ms
;     2          10ms     10ms
;     3          20ms     15ms
;
STPSPD  EQU      1

NUMERR  EQU      ERROUT-ERRIN

CR      EQU      0DH
LF      EQU      0AH

CODE    SEGMENT
ASSUME  CS:CODE,DS:NOTHING,ES:NOTHING,SS:NOTHING
;---------------------------------------------------------
;
;       DEVICE HEADER
;
DRVDEV  LABEL    WORD
        DW       -1,-1
        DW       0000     ;IBM format-compatible, Block
        DW       STRATEGY
        DW       DRV$IN
DRVMAX  DB       4

DRVTBL  LABEL    WORD
        DW       DRV$INIT
        DW       MEDIA$CHK
        DW       GET$BPB
        DW       CMDERR
        DW       DRV$READ
        DW       EXIT
        DW       EXIT
        DW       EXIT
        DW       DRV$WRIT
        DW       DRV$WRIT
        DW       EXIT
        DW       EXIT
        DW       EXIT

;------------------------------------
;
;       STRATEGY

PTRSAV  DD       0

STRATP  PROC     FAR
STRATEGY:
        MOV      WORD PTR [PTRSAV],BX
        MOV      WORD PTR [PTRSAV+2],ES
        RET
STRATP  ENDP

;--------------------------------------
;
;       MAIN ENTRY
```

```
CMDLEN  =       0       ;LENGTH OF THIS COMMAND
UNIT    =       1       ;SUB UNIT SPECIFIER
CMDC    =       2       ;COMMAND CODE
STATUS  =       3       ;STATUS
MEDIA   =       13      ;MEDIA DESCRIPTOR
TRANS   =       14      ;TRANSFER ADDRESS
COUNT   =       18      ;COUNT OF BLOCKS OR CHARACTERS
START   =       20      ;FIRST BLOCK TO TRANSFER

DRV$IN:
        PUSH    SI
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        PUSH    BX

        LDS     BX,[PTRSAV]     ;GET POINTER TO I/O PACKET

        MOV     AL,BYTE PTR [BX].UNIT   ;AL = UNIT CODE
        MOV     AH,BYTE PTR [BX].MEDIA  ;AH = MEDIA DESCRIP
        MOV     CX,WORD PTR [BX].COUNT  ;CX = COUNT
        MOV     DX,WORD PTR [BX].START  ;DX = START SECTOR
        PUSH    AX
        MOV     AL,BYTE PTR [BX].CMDC   ;Command code
        CMP     AL,11
        JA      CMDERRP                 ;Bad command
        CBW
        SHL     AX,1                    ;2 times command =
                                        ;word table index
        MOV     SI,OFFSET DRVTBL
        ADD     SI,AX                   ;Index into table
        POP     AX                      ;Get back media
                                        ;and unit

        LES     DI,DWORD PTR [BX].TRANS ;ES:DI = TRANSFER
                                        ;ADDRESS

        PUSH    CS
        POP     DS

ASSUME  DS:CODE

        JMP     WORD PTR [SI]                   ;GO DO COMMAND

;----------------------------------------------------------
;
;       EXIT - ALL ROUTINES RETURN THROUGH THIS PATH
;
ASSUME  DS:NOTHING
CMDERRP:
        POP     AX                      ;Clean stack
CMDERR:
        MOV     AL,3                    ;UNKNOWN COMMAND ERROR
        JMP     SHORT ERR$EXIT

ERR$CNT:LDS     BX,[PTRSAV]
        SUB     WORD PTR [BX].COUNT,CX ;# OF SUCCESS. I/Os

ERR$EXIT:
;AL has error code
        MOV     AH,10000001B               ;MARK ERROR RETURN
        JMP     SHORT ERR1

EXITP   PROC    FAR

EXIT:   MOV     AH,00000001B
ERR1:   LDS     BX,[PTRSAV]
        MOV     WORD PTR [BX].STATUS,AX
                                       ;MARK OPERATION COMPLETE

        POP     BX
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        POP     SI
        RET                            ;RESTORE REGS AND RETURN
EXITP   ENDP

CURDRV  DB      -1

TRKTAB  DB      -1,-1,-1,-1

SECCNT  DW      0

DRVLIM  =       8       ;Number of sectors on device
SECLIM  =       13      ;MAXIMUM SECTOR
HDLIM   =       15      ;MAXIMUM HEAD

;WARNING - preserve order of drive and curhd!

DRIVE   DB      0       ;PHYSICAL DRIVE CODE
CURHD   DB      0       ;CURRENT HEAD
CURSEC  DB      0       ;CURRENT SECTOR
CURTRK  DW      0       ;CURRENT TRACK


;
MEDIA$CHK:              ;Always indicates Don't know
ASSUME  DS:CODE
        TEST    AH,00000100B          ;TEST IF MEDIA REMOVABLE
        JZ      MEDIA$EXT
        XOR     DI,DI                   ;SAY I DON'T KNOW
MEDIA$EXT:
        LDS     BX,[PTRSAV]
        MOV     WORD PTR [BX].TRANS,DI
        JMP     EXIT

BUILD$BPB:
ASSUME  DS:CODE
        MOV     AH,BYTE PTR ES:[DI]             ;GET FAT ID BYTE
        CALL    GETBP                           ;TRANSLATE
SETBPB: LDS     BX,[PTRSAV]
        MOV     [BX].MEDIA,AH
        MOV     [BX].COUNT,DI
        MOV     [BX].COUNT+2,CS
        JMP     EXIT

BUILDBP:
ASSUME  DS:NOTHING
;AH is media byte on entry
;DI points to correct BPB on return
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    BX
        MOV     CL,AH           ;SAVE MEDIA
        AND     CL,0F8H         ;NORMALIZE
        CMP     CL,0F8H         ;COMPARE WITH GOOD MEDIA BYTE
        JZ      GOODID
        MOV     AH,0FEH         ;DEFAULT TO 8-SECTOR,
                                ;SINGLE-SIDED
GOODID:
        MOV     AL,1            ;SET NUMBER OF FAT SECTORS
        MOV     BX,64*256+8 ;SET DIR ENTRIES AND SECTOR MAX
        MOV     CX,40*8         ;SET SIZE OF DRIVE
        MOV     DX,01*256+1 ;SET HEAD LIMIT & SEC/ALL UNIT
        MOV     DI,OFFSET DRVBPB
        TEST    AH,00000010B ;TEST FOR 8 OR 9 SECTOR
        JNZ     HAS8            ;NZ = HAS 8 SECTORS
        INC     AL              ;INC NUMBER OF FAT SECTORS
        INC     BL              ;INC SECTOR MAX
        ADD     CX,40           ;INCREASE SIZE
HAS8:   TEST    AH,00000001B       ;TEST FOR 1 OR 2 HEADS
        JZ      HAS1            ;Z = 1 HEAD
        ADD     CX,CX           ;DOUBLE SIZE OF DISK
        MOV     BH,112          ;INCREASE # OF DIREC. ENTRIES
        INC     DH              ;INC SEC/ALL UNIT
        INC     DL              ;INC HEAD LIMIT
HAS1:   MOV     BYTE PTR [DI].2,DH
        MOV     BYTE PTR [DI].6,BH
        MOV     WORD PTR [DI].8,CX
        MOV     BYTE PTR [DI].10,AH
        MOV     BYTE PTR [DI].11,AL
        MOV     BYTE PTR [DI].13,BL
        MOV     BYTE PTR [DI].15,DL
        POP     BX
        POP     DX
        POP     CX
        POP     AX
        RET


;-------------------------------------------------------------
;
;       DISK I/O HANDLERS
;
;ENTRY:
;       AL = DRIVE NUMBER (0-3)
;       AH = MEDIA DESCRIPTOR
;       CX = SECTOR COUNT
;       DX = FIRST SECTOR
;       DS = CS
;       ES:DI = TRANSFER ADDRESS
;EXIT:
;       IF SUCCESSFUL CARRY FLAG = 0
;         ELSE CF=1 AND AL CONTAINS (MS-DOS) ERROR CODE,
          CX # sectors NOT transferred

DRV$READ:
ASSUME  DS:CODE
        JCXZ    DSKOK
        CALL    SETUP
        JC      DSK$IO
        CALL    DISKRD
        JMP     SHORT DSK$IO

DRV$WRIT:
ASSUME  DS:CODE
        JCXZ    DSKOK
        CALL    SETUP
        JC      DSK$IO
        CALL    DISKWRT
ASSUME  DS:NOTHING
DSK$IO: JNC     DSKOK
        JMP     ERR$CNT
DSKOK:  JMP     EXIT


SETUP:
ASSUME  DS:CODE
;Input same as above
;On output
; ES:DI = Trans addr
; DS:BX Points to BPB
; Carry set if error (AL is error code (MS-DOS))
; else
;       [DRIVE] = Drive number (0-3)
;       [SECCNT] = Sectors to transfer
;       [CURSEC] = Sector number of start of I/O
;       [CURHD]  = Head number of start of I/O    ;Set
;       [CURTRK] = Track # of start of I/O ;Seek performed
; All other registers destroyed

        XCHG   BX,DI              ;ES:BX = TRANSFER ADDRESS
        CALL   GETBP              ;DS:DI = PTR TO B.P.B
        MOV    SI,CX
        ADD    SI,DX
        CMP    SI,WORD PTR [DI].DRVLIM
                                 ;COMPARE AGAINST DRIVE MAX
        JBE    INRANGE
        MOV    AL,8
        STC
        RET

INRANGE:
        MOV    [DRIVE],AL
        MOV    [SECCNT],CX     ;SAVE SECTOR COUNT
        XCHG   AX,DX           ;SET UP LOGICAL SECTOR
                               ;FOR DIVIDE
        XOR    DX,DX
        DIV    WORD PTR [DI].SECLIM ;DIVIDE BY SEC PER TRACK
        INC    DL
        MOV    [CURSEC],DL          ;SAVE CURRENT SECTOR
        MOV    CX,WORD PTR [DI].HDLIM ;GET NUMBER OF HEADS
        XOR    DX,DX   ;DIVIDE TRACKS BY HEADS PER CYLINDER
        DIV    CX
        MOV    [CURHD],DL      ;SAVE CURRENT HEAD
        MOV    [CURTRK],AX     ;SAVE CURRENT TRACK
SEEK:
        PUSH   BX              ;Xaddr
        PUSH   DI              ;BPB pointer
        CALL   CHKNEW          ;Unload head if change drives
        CALL   DRIVESEL
        MOV    BL,[DRIVE]
        XOR    BH,BH           ;BX drive index
        ADD    BX,OFFSET TRKTAB        ;Get current track
        MOV    AX,[CURTRK]
        MOV    DL,AL         ;Save desired track
        XCHG   AL,DS:[BX]    ;Make desired track current
        OUT    DISK+1,AL     ;Tell Controller current track
        CMP    AL,DL         ;At correct track?
        JZ     SEEKRET       ;Done if yes
        MOV    BH,2          ;Seek retry count
        CMP    AL,-1         ;Position Known?
        JNZ    NOHOME        ;If not home head
TRYSK:
        CALL   HOME
        JC     SEEKERR
NOHOME:
        MOV    AL,DL
        OUT    DISK+3,AL       ;Desired track
        MOV    AL,1CH+STPSPD   ;Seek
        CALL   DCOM
        AND    AL,98H    ;Accept not rdy, seek, & CRC errors
        JZ     SEEKRET
        JS     SEEKERR         ;No retries if not ready
        DEC     BH
        JNZ     TRYSK
SEEKERR:
        MOV     BL,[DRIVE]
        XOR     BH,BH           ;BX drive index
        ADD     BX,OFFSET TRKTAB        ;Get current track
        MOV     BYTE PTR DS: BX ,-1     ;Make current track
                                        ;lunknown
        CALL    GETERRCD
        MOV     CX,[SECCNT]     ;Nothing transferred
        POP     BX              ;BPB pointer
        POP     DI              ;Xaddr
        RET

SEEKRET:
        POP     BX              ;BPB pointer
        POP     DI              ;Xaddr
        CLC
        RET

;-----------------------------------------------
;
;       READ
;

DISKRD:
ASSUME  DS:CODE
        MOV     CX,[SECCNT]
RDLP:
        CALL    PRESET
        PUSH    BX
        MOV     BL,10              ;Retry count
        MOV     DX,DISK+3          ;Data port
RDAGN:
        MOV     AL,80H             ;Read command
        CLI                        ;Disable for 1793
        OUT     DISK,AL            ;Output read command
        MOV     BP,DI              ;Save address for retry
        JMP     SHORT RLOOPENTRY
RLOOP:
        STOSB
RLOOPENTRY:
        IN      AL,DISK+5          ;Wait for DRQ or INTRQ
        SHR     AL,1
        IN      AL,DX              ;Read data
        JNC     RLOOP
        STI                        ;Ints OK now
        CALL    GETSTAT
        AND     AL,9CH
        JZ      RDPOP              ;Ok
        MOV     DI,BP              ;Get back transfer
        DEC     BL
        JNZ     RDAGN
        CMP     AL,10H             ;Record not found?
        JNZ     GOT_CODE           ;No
        MOV     AL,1                    ;Map it
GOT_CODE:
        CALL    GETERRCD
        POP     BX
        RET

RDPOP:
        POP     BX
        LOOP    RDLP
        CLC
        RET


;----------------------------------------------
;
;       WRITE
;

DISKWRT:
ASSUME  DS:CODE
        MOV     CX,[SECCNT]
        MOV     SI,DI
        PUSH    ES
        POP     DS
ASSUME  DS:NOTHING
WRLP:
        CALL    PRESET
        PUSH    BX
        MOV     BL,10                   ;Retry count
        MOV     DX,DISK+3               ;Data port
WRAGN:
        MOV     AL,0A0H            ;Write command
        CLI                        ;Disable for 1793
        OUT     DISK,AL            ;Output write command
        MOV     BP,SI              ;Save address for retry
WRLOOP:
        IN      AL,DISK+5
        SHR     AL,1
        LODSB                      ;Get data
        OUT     DX,AL              ;Write data
        JNC     WRLOOP
        STI                        ;Ints OK now
        DEC     SI
        CALL    GETSTAT
        AND     AL,0FCH
        JZ      WRPOP              ;Ok
        MOV     SI,BP              ;Get back transfer
        DEC     BL
        JNZ     WRAGN
        CALL    GETERRCD
        POP     BX
        RET

WRPOP:
        POP     BX
        LOOP    WRLP
        CLC
        RET


PRESET:
ASSUME  DS:NOTHING
        MOV     AL,[CURSEC]
        CMP     AL,CS:[BX].SECLIM
        JBE     GOTSEC
        MOV     DH,[CURHD]
        INC     DH
        CMP     DH,CS:[BX].HDLIM
        JB      SETHEAD                 ;Select new head
        CALL    STEP                    ;Go on to next track
        XOR     DH,DH                   ;Select head zero
SETHEAD:
        MOV     [CURHD],DH
        CALL    DRIVESEL
        MOV     AL,1                    ;First sector
        MOV     [CURSEC],AL             ;Reset CURSEC
GOTSEC:
        OUT     DISK+2,AL       ;Tell controller which sector
        INC     [CURSEC]        ;We go on to next sector
        RET

STEP:
ASSUME  DS:NOTHING
        MOV     AL,58H+STPSPD  ;Step in w/ update, no verify
        CALL    DCOM
        PUSH    BX
        MOV     BL,[DRIVE]
        XOR     BH,BH            ;BX drive index
        ADD     BX,OFFSET TRKTAB        ;Get current track
        INC     BYTE PTR CS:[BX]        ;Next track
        POP     BX
        RET

HOME:
ASSUME  DS:NOTHING
        MOV     BL,3
TRYHOM:
        MOV     AL,0CH+STPSPD    ;Restore with verify
        CALL    DCOM
        AND     AL,98H
        JZ      RET3
        JS      HOMERR           ;No retries if not ready
        PUSH    AX               ;Save real error code
        MOV     AL,58H+STPSPD    ;Step in w/ update no verify
        CALL    DCOM
        DEC     BL
        POP     AX               ;Get back real error code
        JNZ     TRYHOM
HOMERR:
        STC
```

```
RET3:   RET


CHKNEW:
ASSUME  DS:NOTHING
        MOV     AL,[DRIVE]      ;Get disk drive number
        MOV     AH,AL
        XCHG    AL,[CURDRV]     ;Make new drive current.
        CMP     AL,AH           ;Changing drives?
        JZ      RET1            ;No
; If changing drives, unload head so the head load delay
;one-shot will fire again. Do it by seeking to the same
;track with the H bit reset.
;
        IN      AL,DISK+1       ;Get current track number
        OUT     DISK+3,AL       ;Make it the track to seek
        MOV     AL,10H          ;Seek and unload head

DCOM:
ASSUME  DS:NOTHING
        OUT     DISK,AL
        PUSH    AX
        AAM                     ;Delay 10 microseconds
        POP     AX
GETSTAT:
        IN      AL,DISK+4
        TEST    AL,DONEBIT
        JZ      GETSTAT
        IN      AL,DISK
RET1:   RET

DRIVESEL:
ASSUME  DS:NOTHING
;Select the drive based on current info
;Only AL altered
        MOV     AL,[DRIVE]
        OR      AL,SMALBIT + DDBIT      ;5 1/4" IBM PC disks
        CMP     [CURHD],0
        JZ      GOTHEAD
        OR      AL,BACKBIT      ;Select side 1
GOTHEAD:
        OUT     DISK+4,AL       ;Select drive and side
        RET

GETERRCD:
ASSUME  DS:NOTHING
        PUSH    CX
        PUSH    ES
        PUSH    DI
        PUSH    CS
        POP     ES              ;Make ES the local segment
        MOV     CS:[LSTERR],AL ;Terminate list w/ error code
        MOV     CX,NUMERR       ;Number of error conditions
        MOV     DI,OFFSET ERRIN ;Point to error conditions
        REPNE   SCASB
        MOV     AL,NUMERR-1[DI] ;Get translation
        STC                     ;Flag error condition
        POP     DI
        POP     ES
        POP     CX
        RET                     ;and return

;*********************************************************
;       BPB FOR AN IBM FLOPPY DISK, VARIOUS PARAMETERS ARE
;       PATCHED BY GETBP TO REFLECT THE TYPE OF MEDIA
;       INSERTED
;       This is a nine sector single side BPB
DRVBPB:
        DW      512             ;Physical sector size in bytes
        DB      1               ;Sectors/allocation unit
        DW      1               ;Reserved sectors for DOS
        DB      2               ;# of allocation tables
        DW      64              ;Number directory entries
        DW      9*40            ;Number 512-byte sectors
        DB      11111100B       ;Media descriptor
        DW      2               ;Number of FAT sectors
        DW      9               ;Sector limit
        DW      1               ;Head limit


INITAB  DW      DRVBPB                  ;Up to four units
        DW      DRVBPB
        DW      DRVBPB
        DW      DRVBPB

ERRIN:  ;DISK ERRORS RETURNED FROM THE 1793 CONTROLER
        DB      80H             ;NO RESPONSE
        DB      40H             ;Write protect
        DB      20H             ;Write Fault
        DB      10H             ;SEEK error
        DB      8               ;CRC error
        DB      1               ;Mapped from 10H
                                ;(record not found) on READ
LSTERR  DB      0               ;ALL OTHER ERRORS

ERROUT: ;RETURNED ERROR CODES CORRESPONDING TO ABOVE
        DB      2               ;NO RESPONSE
        DB      0               ;WRITE ATTEMPT
                                ;ON WRITE-PROTECT DISK
        DB      0AH             ;WRITE FAULT
        DB      6               ;SEEK FAILURE
        DB      4               ;BAD CRC
        DB      8               ;SECTOR NOT FOUND
        DB      12              ;GENERAL ERROR


DRV$INIT:
;
; Determine number of physical drives by reading CONFIG.SYS
;
```

```
ASSUME  DS:CODE
        PUSH    DS
        LDS     SI,[PTRSAV]
ASSUME  DS:NOTHING
        LDS     SI,DWORD PTR [SI.COUNT] ;DS:SI points to
                                        ;CONFIG.SYS
SCAN_LOOP:
        CALL    SCAN_SWITCH
        MOV     AL,CL
        OR      AL,AL
        JZ      SCAN4
        CMP     AL,"s"
        JZ      SCAN4

WERROR: POP     DS
ASSUME  DS:CODE
        MOV     DX,OFFSET ERRMSG2
WERROR2: MOV    AH,9
        INT     21H
        XOR     AX,AX
        PUSH    AX                      ;No units
        JMP     SHORT ABORT

BADNDRV:
        POP     DS
        MOV     DX,OFFSET ERRMSG1
        JMP     WERROR2

SCAN4:
ASSUME  DS:NOTHING
;BX is number of floppies
        OR      BX,BX
        JZ      BADNDRV                 ;User error
        CMP     BX,4
        JA      BADNDRV                 ;User error
        POP     DS
ASSUME  DS:CODE
        PUSH    BX                      ;Save unit count
ABORT:  LDS     BX,[PTRSAV]
ASSUME  DS:NOTHING
        POP     AX
        MOV     BYTE PTR [BX].MEDIA,AL          ;Unit count
        MOV     [DRVMAX],AL
        MOV     WORD PTR [BX].TRANS,OFFSET DRV$INIT ;SET
                                             ;BREAK ADDRESS
        MOV     [BX].TRANS+2,CS
        MOV     WORD PTR [BX].COUNT,OFFSET INITAB
                                  ;SET POINTER TO BPB ARRAY
        MOV     [BX].COUNT+2,CS
        JMP     EXIT
;
; PUT SWITCH IN CL, VALUE IN BX
;
SCAN_SWITCH:
        XOR     BX,BX
        MOV     CX,BX
        LODSB
        CMP     AL,10
        JZ      NUMRET
        CMP     AL,"-"
        JZ      GOT_SWITCH
        CMP     AL,"/"
        JNZ     SCAN_SWITCH
GOT_SWITCH:
        CMP     BYTE PTR [SI+1],":"
        JNZ     TERROR
        LODSB
        OR      AL,20H              ; CONVERT TO LOWER CASE
        MOV     CL,AL               ; GET SWITCH
        LODSB                       ; SKIP ":"
;
;   GET NUMBER POINTED TO BY [SI]
;
;   WIPES OUT AX,DX ONLY     BX RETURNS NUMBER
;
GETNUM1:LODSB
        SUB     AL,"0"
        JB      CHKRET
        CMP     AL,9
        JA      CHKRET
        CBW
        XCHG    AX,BX
        MOV     DX,10
        MUL     DX
        ADD     BX,AX
        JMP     GETNUM1

CHKRET: ADD     AL,"0"
        CMP     AL," "
        JBE     NUMRET
        CMP     AL,"-"
        JZ      NUMRET
        CMP     AL,"/"
        JZ      NUMRET
TERROR:
        POP     DS                  ; GET RID OF RETURN ADDRESS
        JMP     WERROR
NUMRET: DEC     SI
        RET

ERRMSG1 DB      "SMLDRV: Bad number of drives",13,10,"$"
ERRMSG2 DB      "SMLDRV: Invalid parameter",13,10,"$"
CODE    ENDS
        END
```

2.8.2 Character Device Driver

The following program illustrates a character device driver program.

```
;******************** A CHARACTER DEVICE ********************

TITLE   VT52 CONSOLE FOR 2.0    (IBM)

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;
;       IBM ADDRESSES FOR I/O
;
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

        CR=13                  ;CARRIAGE RETURN
        BACKSP=8               ;BACKSPACE
        ESC=1BH
        BRKADR=6CH             ;006C  BREAK VECTOR ADDRESS
        ASNMAX=200             ;SIZE OF KEY ASSIGNMENT BUFFER

CODE    SEGMENT BYTE

   ASSUME CS:CODE,DS:NOTHING,ES:NOTHING
;----------------------------------------------------------
;
;       C O N - CONSOLE DEVICE DRIVER
;
CONDEV:                                ;HEADER FOR DEVICE "CON"
        DW      -1,-1
        DW      1000000000010011B  ;CON IN AND CON OUT
        DW      STRATEGY
        DW      ENTRY
        DB      'CON     '

;-----------------------------------------------------------
;
;       COMMAND JUMP TABLES
CONTBL:
        DW      CON$INIT
        DW      EXIT
        DW      EXIT
        DW      CMDERR
        DW      CON$READ
        DW      CON$RDND
        DW      EXIT
        DW      CON$FLSH
        DW      CON$WRIT
        DW      CON$WRIT
        DW      EXIT
        DW      EXIT

CMDTABL DB      'A'
```

```
        DW      CUU             ;cursor up
        DB      'B'
        DW      CUD             ;cursor down
        DB      'C'
        DW      CUF             ;cursor forward
        DB      'D'
        DW      CUB             ;cursor back
        DB      'H'
        DW      CUH             ;cursor position
        DB      'J'
        DW      ED              ;erase display
        DB      'K'
        DW      EL              ;erase line
        DB      'Y'
        DW      CUP             ;cursor position
        DB      'j'
        DW      PSCP            ;save cursor position
        DB      'k'
        DW      PRCP            ;restore cursor position
        DB      'y'
        DW      RM              ;reset mode
        DB      'x'
        DW      SM              ;set mode
        DB      00

PAGE
;---------------------------------------------------
;
;       Device entry point
;
CMDLEN  =       0       ;LENGTH OF THIS COMMAND
UNIT    =       1       ;SUB UNIT SPECIFIER
CMD     =       2       ;COMMAND CODE
STATUS  =       3       ;STATUS
MEDIA   =       13      ;MEDIA DESCRIPTOR
TRANS   =       14      ;TRANSFER ADDRESS
COUNT   =       18      ;COUNT OF BLOCKS OR CHARACTERS
START   =       20      ;FIRST BLOCK TO TRANSFER

PTRSAV  DD      0

STRATP  PROC    FAR

STRATEGY:
        MOV     WORD PTR CS:]PTRSAV[,BX
        MOV     WORD PTR CS:]PTRSAV+2[,ES
        RET

STRATP  ENDP

ENTRY:
        PUSH    SI
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    DI
        PUSH    BP
        PUSH    DS
        PUSH    ES
        PUSH    BX

        LDS     BX,CS:[PTRSAV]  ;GET POINTER TO I/O PACKET

        MOV     CX,WORD PTR DS:[BX].COUNT    ;CX = COUNT

        MOV     AL,BYTE PTR DS:[BX].CMD
        CBW
        MOV     SI,OFFSET CONTBL
        ADD     SI,AX
        ADD     SI,AX
        CMP     AL,11
        JA      CMDERR

        LES     DI,DWORD PTR DS:[BX].TRANS

        PUSH    CS
        POP     DS

        ASSUME  DS:CODE

        JMP     WORD PTR [SI]               ;GO DO COMMAND

PAGE
;=====================================================
;=
;=      SUBROUTINES SHARED BY MULTIPLE DEVICES
;=
;=====================================================
;-----------------------------------------------------
;
;       EXIT - ALL ROUTINES RETURN THROUGH THIS PATH
;
BUS$EXIT:                               ;DEVICE BUSY EXIT
        MOV     AH,00000011B
        JMP     SHORT ERR1

CMDERR:
        MOV     AL,3                ;UNKNOWN COMMAND ERROR

ERR$EXIT:
        MOV     AH,10000001B            ;MARK ERROR RETURN
        JMP     SHORT ERR1

EXITP   PROC    FAR

EXIT:   MOV     AH,00000001B
ERR1:   LDS     BX,CS:[PTRSAV]
        MOV     WORD PTR [BX].STATUS,AX ;MARK
                                        ;OPERATION COMPLETE
        POP     BX
        POP     ES
        POP     DS
        POP     BP
        POP     DI
        POP     DX
        POP     CX
        POP     AX
        POP     SI
        RET                             ;RESTORE REGS AND RETURN
EXITP   ENDP
;-----------------------------------------------
;
;       BREAK KEY HANDLING
;
BREAK:
        MOV     CS:ALTAH,3              ;INDICATE BREAK KEY SET
INTRET: IRET

PAGE
;
;       WARNING - Variables are very order dependent,
;                 so be careful when adding new ones!
;
WRAP    DB      0                   ; 0 = WRAP, 1 = NO WRAP
STATE   DW      S1
MODE    DB      3
MAXCOL  DB      79
COL     DB      0
ROW     DB      0
SAVCR   DW      0
ALTAH   DB      0                   ;Special key handling

;---------------------------------------------------------
;
;    CHROUT - WRITE OUT CHAR IN AL USING CURRENT ATTRIBUTE
;
ATTRW   LABEL   WORD
ATTR    DB      00000111B           ;CHARACTER ATTRIBUTE
BPAGE   DB      0                   ;BASE PAGE
base    dw      0b800h

chrout: cmp     al,13
        jnz     trylf
        mov     [col],0
        jmp     short setit

trylf:  cmp     al,10
        jz      lf
        cmp     al,7
        jnz     tryback
torom:
        mov     bx,[attrw]
        and     bl,7
        mov     ah,14
        int     10h
ret5:   ret

tryback:
        cmp     al,8
        jnz     outchr
        cmp     [col],0
        jz      ret5
        dec     [col]
        jmp     short setit

outchr:
        mov     bx,[attrw]
        mov     cx,1
        mov     ah,9
        int     10h
        inc     [col]
        mov     al,[col]
        cmp     al,[maxcol]
        jbe     setit
        cmp     [wrap],0
        jz      outchr1
        dec     [col]
        ret
outchr1:
        mov     [col],0
lf:     inc     [row]
        cmp     [row],24
        jb      setit
        mov     [row],23
        call    scroll

setit:  mov     dh,row
        mov     dl,col
        xor     bh,bh
        mov     ah,2
        int     10h
        ret

scroll: call    getmod
        cmp     al,2
        jz      myscroll
        cmp     al,3
        jz      myscroll
        mov     al,10
        jmp     torom
myscroll:
        mov     bh,[attr]
        mov     bl,' '
        mov     bp,80
        mov     ax,[base]
        mov     es,ax
        mov     ds,ax
        xor     di,di
        mov     si,160
        mov     cx,23*80
        cld
        cmp     ax,0b800h
        jz      colorcard

        rep     movsw
        mov     ax,bx
        mov     cx,bp
        rep     stosw
sret:   push    cs
        pop     ds
        ret

colorcard:
        mov     dx,3dah
wait2:  in      al,dx
        test    al,8
        jz      wait2
        mov     al,25h
        mov     dx,3d8h
        out     dx,al                   ;turn off video
        rep     movsw
        mov     ax,bx
        mov     cx,bp
        rep     stosw
        mov     al,29h
        mov     dx,3d8h
        out     dx,al                   ;turn on video
        jmp     sret

GETMOD: MOV     AH,15
        INT     16                      ;get column information
        MOV     BPAGE,BH
        DEC     AH
        MOV     WORD PTR MODE,AX
        RET
;-------------------------------------------------------
;
;       CONSOLE READ ROUTINE
;
CON$READ:
        JCXZ    CON$EXIT
CON$LOOP:
        PUSH    CX                      ;SAVE COUNT
        CALL    CHRIN                   ;GET CHAR IN AL
        POP     CX
        STOSB                           ;STORE CHAR AT ES:DI
        LOOP    CON$LOOP
CON$EXIT:
        JMP     EXIT
;----------------------------------------------------------
;
;       INPUT SINGLE CHAR INTO AL
;
CHRIN:  XOR     AX,AX
```

```
        XCHG    AL,ALTAH        ;GET CHARACTER & ZERO ALTAH
        OR      AL,AL
        JNZ     KEYRET

INAGN:  XOR     AH,AH
        INT     22
ALT10:
        OR      AX,AX          ;Check for non-key after BREAK
        JZ      INAGN
        OR      AL,AL          ;SPECIAL CASE?
        JNZ     KEYRET
        MOV     ALTAH,AH           ;STORE SPECIAL KEY
KEYRET: RET
;-----------------------------------------------------------
;
;       KEYBOARD NON DESTRUCTIVE READ, NO WAIT
;
CON$RDND:
        MOV     AL,[ALTAH]
        OR      AL,AL
        JNZ     RDEXIT

RD1:    MOV     AH,1
        INT     22
        JZ      CONBUS
        OR      AX,AX
        JNZ     RDEXIT
        MOV     AH,0
        INT     22
        JMP     CON$RDND

RDEXIT: LDS     BX,[PTRSAV]
        MOV     [BX].MEDIA,AL
EXVEC:  JMP     EXIT
CONBUS: JMP     BUS$EXIT
;-----------------------------------------------------------
;
;       KEYBOARD FLUSH ROUTINE
;
CON$FLSH:
        MOV     [ALTAH],0           ;Clear out holding buffer

        PUSH    DS
        XOR     BP,BP
        MOV     DS,BP                       ;Select segment 0
        MOV     DS:BYTE PTR 41AH,1EH        ;Reset KB queue head
                                            ;pointer
        MOV     DS:BYTE PTR 41CH,1EH        ;Reset tail pointer
        POP     DS
        JMP     EXVEC
;-----------------------------------------------------------
;
;       CONSOLE WRITE ROUTINE
;
CON$WRIT:
        JCXZ    EXVEC
        PUSH    CX
        MOV     AH,3                ;SET CURRENT CURSOR POSITION
        XOR     BX,BX
        INT     16
        MOV     WORD PTR [COL],DX
        POP     CX

CON$LP: MOV     AL,ES:[DI]          ;GET CHAR
        INC     DI
        CALL    OUTC                ;OUTPUT CHAR
        LOOP    CON$LP              ;REPEAT UNTIL ALL THROUGH
        JMP     EXVEC

COUT:   STI
        PUSH    DS
        PUSH    CS
        POP     DS
        CALL    OUTC
        POP     DS
        IRET

OUTC:   PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    ES
        PUSH    BP
        CALL    VIDEO
        POP     BP
        POP     ES
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET


;----------------------------------------------------------
;
;       OUTPUT SINGLE CHAR IN AL TO VIDEO DEVICE
;
VIDEO:  MOV     SI,OFFSET STATE
        JMP     [SI]

S1:     CMP     AL,ESC                      ;ESCAPE SEQUENCE?
        JNZ     S1B
        MOV     WORD PTR [SI],OFFSET S2
        RET

S1B:    CALL    CHROUT
S1A:    MOV     WORD PTR [STATE],OFFSET S1
        RET
```

```
S2:      PUSH    AX
         CALL    GETMOD
         POP     AX
         MOV     BX,OFFSET CMDTABL-3
S7A:     ADD     BX,3
         CMP     BYTE PTR [BX],0
         JZ      S1A
         CMP     BYTE PTR [BX],AL
         JNZ     S7A
         JMP     WORD PTR [BX+1]


MOVCUR:  CMP     BYTE PTR [BX],AH
         JZ      SETCUR
         ADD     BYTE PTR [BX],AL
SETCUR:  MOV     DX,WORD PTR COL
         XOR     BX,BX
         MOV     AH,2
         INT     16
         JMP     S1A


CUP:     MOV     WORD PTR [SI],OFFSET CUP1
         RET
CUP1:    SUB     AL,32
         MOV     BYTE PTR [ROW],AL
         MOV     WORD PTR [SI],OFFSET CUP2
         RET
CUP2:    SUB     AL,32
         MOV     BYTE PTR [COL],AL
         JMP     SETCUR

SM:      MOV     WORD PTR [SI],OFFSET S1A
         RET


CUH:     MOV     WORD PTR COL,0
         JMP     SETCUR

CUF:     MOV     AH,MAXCOL
         MOV     AL,1
CUF1:    MOV     BX,OFFSET COL
         JMP     MOVCUR

CUB:     MOV     AX,00FFH
         JMP     CUF1

CUU:     MOV     AX,00FFH
CUU1:    MOV     BX,OFFSET ROW
         JMP     MOVCUR

CUD:     MOV     AX,23*256+1
         JMP     CUU1
```

```
PSCP:   MOV     AX,WORD PTR COL
        MOV     SAVCR,AX
        JMP     SETCUR

PRCP:   MOV     AX,SAVCR
        MOV     WORD PTR COL,AX
        JMP     SETCUR

ED:     CMP     BYTE PTR [ROW],24
        JAE     EL1

        MOV     CX,WORD PTR COL
        MOV     DH,24
        JMP     ERASE

EL1:    MOV     BYTE PTR [COL],0
EL:     MOV     CX,WORD PTR [COL]
EL2:    MOV     DH,CH
ERASE:  MOV     DL,MAXCOL
        MOV     BH,ATTR
        MOV     AX,0600H
        INT     16
ED3:    JMP     SETCUR


RM:     MOV     WORD PTR [SI],OFFSET RM1
        RET
RM1:    XOR     CX,CX
        MOV     CH,24
        JMP     EL2

CON$INIT:
        int     11h
        and     al,00110000b
        cmp     al,00110000b
        jnz     iscolor
        mov     [base],0b000h           ;look for bw card
iscolor:
        cmp     al,00010000b            ;look for 40 col mode
        ja      setbrk
        mov     [mode],0
        mov     [maxcol],39

setbrk:
        XOR     BX,BX
        MOV     DS,BX
        MOV     BX,BRKADR
        MOV     WORD PTR [BX],OFFSET BREAK
        MOV     WORD PTR [BX+2],CS

        MOV     BX,29H*4
        MOV     WORD PTR [BX],OFFSET COUT
        MOV     WORD PTR [BX+2],CS
        LDS     BX,CS:[PTRSAV]
        MOV     WORD PTR [BX].TRANS,OFFSET CON$INIT
                              ;SET BREAK ADDRESS
        MOV     [BX].TRANS+2,CS
        JMP     EXIT

CODE    ENDS
        END
```

CHAPTER 3

MS-DOS TECHNICAL INFORMATION

3.1 MS-DOS INITIALIZATION

MS-DOS initialization consists of several steps. Typically, a ROM (Read Only Memory) bootstrap obtains control, and then reads the boot sector off the disk. The boot sector then reads the following files:

IO.SYS
MSDOS.SYS

Once these files are read, the boot process begins.

3.2 THE COMMAND PROCESSOR

The command processor supplied with MS-DOS (file COMMAND.COM.) consists of 3 parts:

1. A resident part resides in memory immediately following MSDOS.SYS and its data area. This part contains routines to process Interrupts 23H (CONTROL-C Exit Address) and 24H (Fatal Error Abort Address), as well as a routine to reload the transient part, if needed. All standard MS-DOS error handling is done within this part of COMMAND.COM. This includes displaying error messages and processing the Abort, Retry, or Ignore messages.

2. An initialization part follows the resident part. During startup, the initialization part is given control; it contains the AUTOEXEC file processor setup routine. The initialization part determines the segment address at which programs can be loaded. It is overlaid by the first program COMMAND.COM loads because it is no longer needed.

3. A transient part is loaded at the high end of memory. This part contains all of the internal command processors and the batch file processor.

   The transient part of the command processor produces the system prompt (such as A>), reads the command from keyboard (or batch file) and causes it to be executed. For external commands, this part builds a command line and issues the EXEC system call (Function Request 4BH) to load and transfer control to the program.

3.3 MS-DOS DISK ALLOCATION

The MS-DOS area is formatted as follows:

Reserved area - variable size
First copy of file allocation table - variable size
Second copy of file allocation table - variable size (optional)
Additional copies of file allocation table - variable size (optional)
Root directory - variable size
File data area

Allocation of space for a file in the data area is not pre-allocated. The space is allocated one cluster at a time. A cluster consists of one or more consecutive sectors; all of the clusters for a file are "chained" together in the File Allocation Table (FAT). (Refer to Section 3.5, "File Allocation Table.") There is usually a second copy of the FAT kept, for consistency. Should the disk develop a bad sector in the middle of the first FAT, the second can be used. This avoids loss of data due to an unusable disk.

3.4 MS-DOS DISK DIRECTORY

FORMAT builds the root directory for all disks. Its location on disk and the maximum number of entries are dependent on the media.

Since directories other than the root directory are regarded as files by MS-DOS, there is no limit to the number of files they may contain.

All directory entries are 32 bytes in length, and are in the following format (note that byte offsets are in hexadecimal):

0-7 Filename. Eight characters, left aligned and padded, if necessary, with blanks. The first byte of this field indicates the file status as follows:

- 00H The directory entry has never been used. This is used to limit the length of directory searches, for performance reasons.
- 2EH The entry is for a directory. If the second byte is also 2EH, then the cluster field contains the cluster number of this directory's parent directory (0000H if the parent directory is the root directory). Otherwise, bytes 01H through 0AH are all spaces, and the cluster field contains the cluster number of this directory.
- E5H The file was used, but it has been erased.

Any other character is the first character of a filename.

8-0A Filename extension.

0B File attribute. The attribute byte is mapped as follows (values are in hexadecimal):

- 01 File is marked read-only. An attempt to open the file for writing using the Open File system call (Function Request 3DH) results in an error code being returned. This value can be used along with other values below. Attempts to delete the file with the Delete File system call (13H) or Delete a Directory Entry (41H) will also fail.
- 02 Hidden file. The file is excluded from normal directory searches.
- 04 System file. The file is excluded from normal directory searches.
- 08 The entry contains the volume label in the first 11 bytes. The entry contains no other usable information

(except date and time of creation), and may exist only in the root directory.

10 The entry defines a sub-directory, and is excluded from normal directory searches.

20 Archive bit. The bit is set to "on" whenever the file has been written to and closed.

Note: The system files (IO.SYS and MSDOS.SYS) are marked as read-only, hidden, and system files. Files can be marked hidden when they are created. Also, the read-only, hidden, system, and archive attributes may be changed through the Change Attributes system call (Function Request 43H).

0C-15 Reserved.

16-17 Time the file was created or last updated. The hour, minutes, and seconds are mapped into two bytes as follows:

```
Offset 17H
| H | H | H | H | H | M | M | M |
  7               3   2

Offset 16H
| M | M | M | S | S | S | S | S |
          5   4               0
```

where:

H	is the binary number of hours (0-23)
M	is the binary number of minutes (0-59)
S	is the binary number of two-second increments

18-19 Date the file was created or last updated. The year, month, and day are mapped into two bytes as follows:

```
Offset 19H
| Y | Y | Y | Y | Y | Y | Y | M |
 7                        1   0

Offset 18H
| M | M | M | D | D | D | D | D |
          5   4               0
```

where:

Y is 0-119 (1980-2099)
M is 1-12
D is 1-31

1A-1B Starting cluster; the cluster number of the first cluster in the file.

Note that the first cluster for data space on all disks is cluster 002.

The cluster number is stored with the least significant byte first.

NOTE

Refer to Section 3.5.1, "How to Use the File Allocation Table," for details about converting cluster numbers to logical sector numbers.

1C-1F File size in bytes. The first word of this four-byte field is the low-order part of the size.

3.5 FILE ALLOCATION TABLE (FAT)

The following information is included for system programmers who wish to write installable device drivers. This section explains how MS-DOS uses the File Allocation Table to convert the clusters of a file to logical sector numbers. The driver is then responsible for locating the logical sector on disk. Programs must use the MS-DOS file management function calls for accessing files; programs that access the FAT are not guaranteed to be upwardly-compatible with future releases of MS-DOS.

The File Allocation Table is an array of 12-bit entries (1.5 bytes) for each cluster on the disk. The first two FAT entries map a portion of the directory; these FAT entries indicate the size and format of the disk.

The second and third bytes currently always contain FFH.

The third FAT entry, which starts at byte offset 4, begins the mapping of the data area (cluster 002). Files in the data area are not always written sequentially on the disk. The data area is allocated one cluster at a time, skipping over clusters already allocated. The first free cluster found will be the next cluster allocated, regardless of its physical location on the disk. This permits the most efficient utilization of disk space because clusters made available by erasing files can be allocated for new files.

Each FAT entry contains three hexadecimal characters:

000	If the cluster is unused and available.
FF7	The cluster has a bad sector in it. MS-DOS will not allocate such a cluster. CHKDSK counts the number of bad clusters for its report. These bad clusters are not part of any allocation chain.
FF8-FFF	Indicates the last cluster of a file.
XXX	Any other characters that are the cluster number of the next cluster in the file. The cluster number of the first cluster in the file is kept in the file's directory entry.

The File Allocation Table always begins on the first section after the reserved sectors. If the FAT is larger than one sector, the sectors are continguous. Two copies of the FAT are usually written for data integrity. The FAT is read into one of the MS-DOS buffers whenever needed (open, read, write, etc.). For performance reasons, this buffer is given a high priority to keep it in memory as long as possible.

3.5.1 How To Use The File Allocation Table

Use the directory entry to find the starting cluster of the file. Next, to locate each subsequent cluster of the file:

1. Multiply the cluster number just used by 1.5 (each FAT entry is 1.5 bytes long).

2. The whole part of the product is an offset into the FAT, pointing to the entry that maps the cluster just used. That entry contains the cluster number of the next cluster of the file.

3. Use a MOV instruction to move the word at the calculated FAT offset into a register.

4. If the last cluster used was an even number, keep the low-order 12 bits of the register by ANDing it with FFF; otherwise, keep the high-order 12 bits by shifting the register right 4 bits with a SHR instruction.

5. If the resultant 12 bits are FF8H-FFFH, the file contains no more clusters. Otherwise, the 12 bits contain the cluster number of the next cluster in the file.

To convert the cluster to a logical sector number (relative sector, such as that used by Interrupts 25H and 26H and by DEBUG):

1. Subtract 2 from the cluster number.

2. Multiply the result by the number of sectors per cluster.

3. Add to this result the logical sector number of the beginning of the data area.

3.6 MS-DOS STANDARD DISK FORMATS

On an MS-DOS disk, the clusters are arranged on disk to minimize head movement for multi-sided media. All of the space on a track (or cylinder) is allocated before moving on to the next track. This is accomplished by using the sequential sectors on the lowest-numbered head, then all the sectors on the next head, and so on until all sectors on all heads of the track are used. The next sector to be used will be sector 1 on head 0 of the next track.

For disks, the following table can be used:

# Sides	Sectors/ Track	FAT size Sectors	Dir Sectors	Dir Entries	Sectors/ Cluster
1	8	1	4	64	1
2	8	1	7	112	2
1	9	2	4	64	1
2	9	2	7	112	2

Figure 4. 5-1/4" Disk Format

The first byte of the FAT can sometimes be used to determine the format of the disk. The following 5-1/4" formats have been defined for the IBM Personal Computer, based on values of the first byte of the FAT. The formats in Table 3.1 are considered to be the standard disk formats for MS-DOS.

Table 3.1 MS-DOS Standard Disk Formats

	5-1/4	5-1/4	5-1/4	5-1/4	8	8	8
No. sides	1	1	2	2	1	1	2
Tracks/side	40	40	40	40	77	77	77
Bytes/ sector	512	512	512	512	128	128	1024
Sectors/ track	8	9	8	9	26	26	8
Sectors/allocation unit	1	1	2	2	4	4	1
Reserved sectors	1	1	1	1	1	4	1
No. FATs	2	2	2	2	2	2	2
Root directory entries	64	64	112	112	68	68	192
No. sectors	320	360	640	720	2002	2002	616
Media Descriptor Byte	FE	FC	FF	FD	FE*	FD	FE*
Sectors for 1 FAT	1	2	1	2	6	6	2

*The two media descriptor bytes that are the same for 8" disks (FEH) is not a misprint. To establish whether a disk is single- or double-density, a read of a single-density address mark should be made. If an error occurs, the media is double-density.

CHAPTER 4

MS-DOS CONTROL BLOCKS AND WORK AREAS

4.1 TYPICAL MS-DOS MEMORY MAP

0000:0000	Interrupt vector table
XXXX:0000	IO.SYS - MS-DOS interface to hardware
XXXX:0000	MSDOS.SYS - MS-DOS interrupt handlers, service routines (Interrupt 21H functions)
	MS-DOS buffers, control areas, and installed device drivers
XXXX:0000	Resident part of COMMAND.COM - Interrupt handlers for Interrupts 22H (Terminate Address), 23H (CONTROL-C Exit Address), 24H (Fatal Error Abort Address) and code to reload the transient part
XXXX:0000	External command or utility - (.COM or .EXE file)
XXXX:0000	User stack for .COM files (256 bytes)
XXXX:0000	Transient part of COMMAND.COM - Command interpreter, internal commands, batch processor

1. Memory map addresses are in segment:offset format. For example, 0090:0000 is absolute address 0900H.

2. User memory is allocated from the lowest end of available memory that will meet the allocation request.

4.2 MS-DOS PROGRAM SEGMENT

When an external command is typed, or when you execute a program through the EXEC system call, MS-DOS determines the lowest available free memory address to use as the start of the program. This area is called the Program Segment.

The first 256 bytes of the Program Segment are set up by the EXEC system call for the program being loaded into memory. The program is then loaded following this block. An .EXE file with minalloc and maxalloc both set to zero is loaded as high as possible.

At offset 0 within the Program Segment, MS-DOS builds the Program Segment Prefix control block. The program returns from EXEC by one of four methods:

1. A long jump to offset 0 in the Program Segment Prefix
2. By issuing an INT 20H with CS:0 pointing at the PSP
3. By issuing an INT 21H with register AH=0 with CS:0 pointing at the PSP, or 4CH and no restrictions on CS
4. By a long call to location 50H in the Program Segment Prefix with AH=0 or Function Request 4CH

NOTE

It is the responsibility of all programs to ensure that the CS register contains the segment address of the Program Segment Prefix when terminating via any of these methods, except Function Request 4CH. For this reason, using Function Request 4CH is the preferred method.

All four methods result in transferring control to the program that issued the EXEC. During this returning process, Interrupts 22H, 23H, and 24H (Terminate Address, CONTROL-C Exit Address, and Fatal Error Abort Address) addresses are restored from the values saved in the Program Segment Prefix of the terminating program. Control is then given to the terminate address. If this is a program returning to COMMAND.COM, control transfers to its resident portion. If a batch file was in process, it is continued;

otherwise, COMMAND.COM performs a checksum on the transient part, reloads it if necessary, then issues the system prompt and waits for you to type the next command.

When a program receives control, the following conditions are in effect:

For all programs:

> The segment address of the passed environment is contained at offset 2CH in the Program Segment Prefix.
>
> The environment is a series of ASCII strings (totaling less than 32K) in the form:
>
> NAME=parameter
>
> Each string is terminated by a byte of zeros, and the set of strings is terminated by another byte of zeros. The environment built by the command processor contains at least a COMSPEC= string (the parameters on COMSPEC define the path used by MS-DOS to locate COMMAND.COM on disk). The last PATH and PROMPT commands issued will also be in the environment, along with any environment strings defined with the MS-DOS SET command.
>
> The environment that is passed is a copy of the invoking process environment. If your application uses a "keep process" concept, you should be aware that the copy of the environment passed to you is static. That is, it will not change even if subsequent SET, PATH, or PROMPT commands are issued.
>
> Offset 50H in the Program Segment Prefix contains code to call the MS-DOS function dispatcher. By placing the desired function request number in AH, a program can issue a far call to offset 50H to invoke an MS-DOS function, rather than issuing an Interrupt 21H. Since this is a call and not an interrupt, MS-DOS may place any code appropriate to making a system call at this position. This makes the process of calling the system portable.
>
> The Disk Transfer Address (DTA) is set to 80H (default DTA in the Program Segment Prefix).

File control blocks at 5CH and 6CH are formatted from the first two parameters typed when the command was entered. If either parameter contained a pathname, then the corresponding FCB contains only the valid drive number. The filename field will not be valid.

An unformatted parameter area at 81H contains all the characters typed after the command (including leading and imbedded delimiters), with the byte at 80H set to the number of characters. If the <, >, or parameters were typed on the command line, they (and the filenames associated with them) will not appear in this area; redirection of standard input and output is transparent to applications.

Offset 6 (one word) contains the number of bytes available in the segment.

Register AX indicates whether or not the drive specifiers (entered with the first two parameters) are valid, as follows:

AL=FF if the first parameter contained an invalid drive specifier (otherwise AL=00)

AH=FF if the second parameter contained an invalid drive specifier (otherwise AH=00)

Offset 2 (one word) contains the segment address of the first byte of unavailable memory. Programs must not modify addresses beyond this point unless they were obtained by allocating memory via the Allocate Memory system call (Function Request 48H).

For Executable (.EXE) programs:

DS and ES registers are set to point to the Program Segment Prefix.

CS,IP,SS, and SP registers are set to the values passed by MS-LINK.

For Executable (.COM) programs:

All four segment registers contain the segment address of the initial allocation block that starts with the Program Segment Prefix control block.

All of user memory is allocated to the program. If the program invokes another program through Function Request 4BH, it must first free some memory through the Set Block (4AH) function call, to provide space for the program being executed.

The Instruction Pointer (IP) is set to 100H.

The Stack Pointer register is set to the end of the program's segment. The segment size at offset 6 is reduced by 100H to allow for a stack of that size.

A word of zeros is placed on top of the stack. This is to allow a user program to exit to COMMAND.COM by doing a RET instruction last. This assumes, however, that the user has maintained his stack and code segments.

Figure 5 illustrates the format of the Program Segment Prefix. All offsets are in hexadecimal.

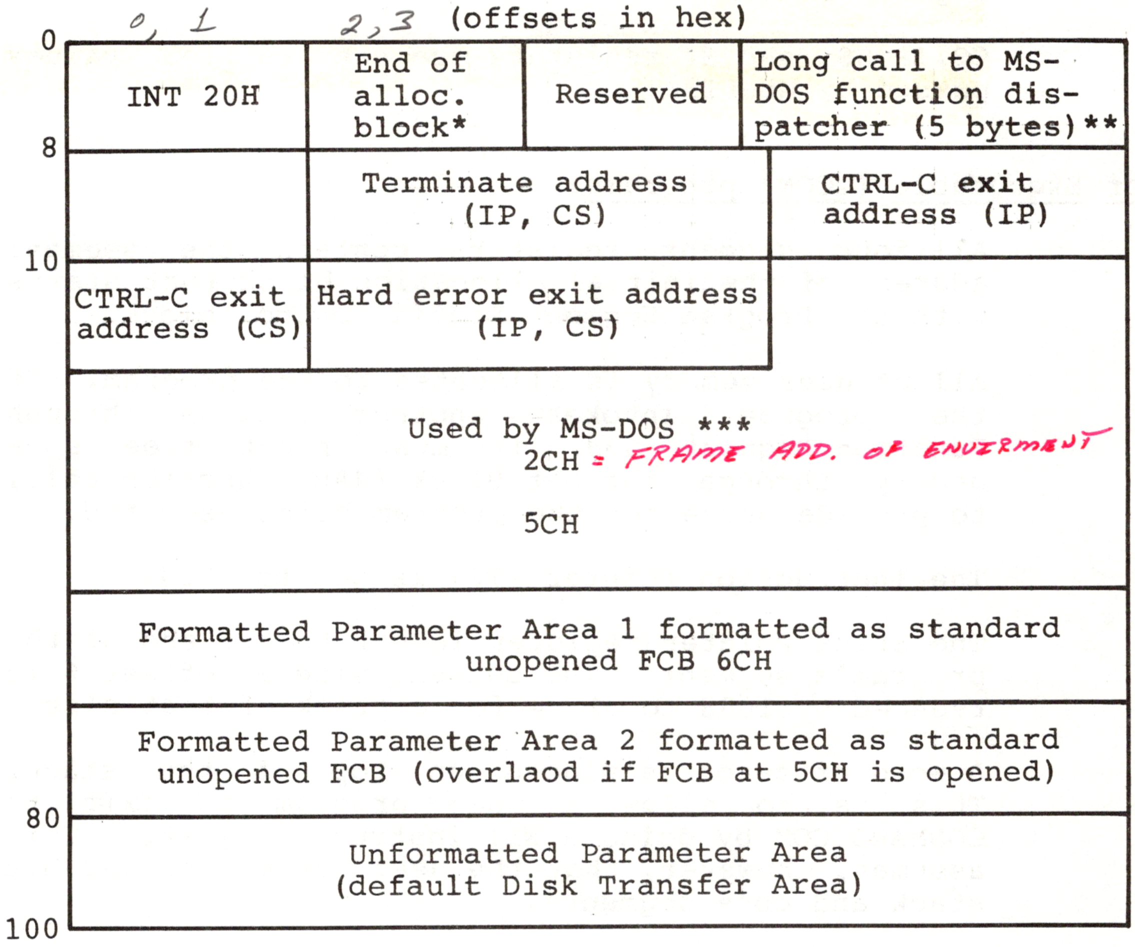

0, 1
2,3
(offsets in hex)
0
INT 20H
End of alloc. block*
Reserved
Long call to MS-DOS function dispatcher (5 bytes)**
8
Terminate address (IP, CS)
CTRL-C exit address (IP)
10
CTRL-C exit address (CS)
Hard error exit address (IP, CS)
Used by MS-DOS ***
2CH = FRAME ADD. OF ENVIRMENT
5CH
Formatted Parameter Area 1 formatted as standard unopened FCB 6CH
Formatted Parameter Area 2 formatted as standard unopened FCB (overlaod if FCB at 5CH is opened)
80
Unformatted Parameter Area (default Disk Transfer Area)
100

Figure 5. Program Segment Prefix

IMPORTANT

Programs must not alter any part of the Program Segment Prefix below offset 5CH.

CHAPTER 5

.EXE FILE STRUCTURE AND LOADING

NOTE

This chapter describes .EXE file structure and loading procedures for systems that use a version of MS-DOS that is lower than 2.0. For MS-DOS 2.0 and higher, use Function Request 4BH, Load and Execute a Program, to load (or load and execute) an .EXE file.

The .EXE files produced by MS-LINK consist of two parts:

Control and relocation information

The load module

The control and relocation information is at the beginning of the file in an area called the header. The load module immediately follows the header.

The header is formatted as follows. (Note that offsets are in hexadecimal.)

Offset	Contents
00-01	Must contain 4DH, 5AH.
02-03	Number of bytes contained in last page; this is useful in reading overlays.
04-05	Size of the file in 512-byte pages, including the header.
06-07	Number of relocation entries in table.

08-09	Size of the header in 16-byte paragraphs. This is used to locate the beginning of the load module in the file.
0A-0B	Minimum number of 16-byte paragraphs required above the end of the loaded program.
0C-0D	Maximum number of 16-byte paragraphs required above the end of the loaded program. If both minalloc and maxalloc are 0, then the program will be loaded as high as possible.
0E-0F	Initial value to be loaded into stack segment before starting program execution. This must be adjusted by relocation.
10-11	Value to be loaded into the SP register before starting program execution.
12-13	Negative sum of all the words in the file.
14-15	Initial value to be loaded into the IP register before starting program execution.
16-17	Initial value to be loaded into the CS register before starting program execution. This must be adjusted by relocation.
18-19	Relative byte offset from beginning of run file to relocation table.
1A-1B	The number of the overlay as generated by MS-LINK.

The relocation table follows the formatted area described above. This table consists of a variable number of relocation items. Each relocation item contains two fields: a two-byte offset value, followed by a two-byte segment value. These two fields contain the offset into the load module of a word which requires modification before the module is given control. The following steps describe this process:

1. The formatted part of the header is read into memory. Its size is 1BH.

2. A portion of memory is allocated depending on the size of the load module and the allocation numbers (0A-0B and 0C-0D). MS-DOS attempts to allocate FFFFH paragraphs. This will always fail, returning the size of the largest free block. If this block is smaller than minalloc and loadsize, then there will be no memory error. If this block is larger than maxalloc and loadsize, MS-DOS will allocate (maxalloc + loadsize). Otherwise, MS-DOS will allocate the largest free block of memory.

3. A Program Segment Prefix is built in the lowest part of the allocated memory.

4. The load module size is calculated by subtracting the header size from the file size. Offsets 04-05 and 08-09 can be used for this calculation. The actual size is downward-adjusted based on the contents of offsets 02-03. Based on the setting of the high/low loader switch, an appropriate segment is determined at which to load the load module. This segment is called the start segment.

5. The load module is read into memory beginning with the start segment.

6. The relocation table items are read into a work area.

7. Each relocation table item segment value is added to the start segment value. This calculated segment, plus the relocation item offset value, points to a word in the load module to which is added the start segment value. The result is placed back into the word in the load module.

8. Once all relocation items have been processed, the SS and SP registers are set from the values in the header. Then, the start segment value is added to SS. The ES and DS registers are set to the segment address of the Program Segment Prefix. The start segment value is added to the header CS register value. The result, along with the header IP value, is the initial CS:IP to transfer to before starting execution of the program.

INDEX

Absolute Disk Read (Interrupt 25H) . . 1-23
Absolute Disk Write (Interrupt 26H) . 1-25
Allocate Memory (Function 48H) 1-128
Archive bit 3-6
ASCIZ 1-107
Attribute field 2-4
Attributes 1-12
AUTOEXEC file 3-2
Auxiliary Input (Function 03H) 1-36
Auxiliary Output (Function 04H) . . . 1-37

Basic 1-1
BIOS 1-25, 2-6
BIOS Parameter Block 2-10, 2-13
Bit 8 2-9
Bit 9 2-9
Block device example 2-20
Block devices 2-1, 2-8, 2-10, 2-16
Boot sector 2-14
BPB 2-10
BPB pointer 2-12
Buffered Keyboard Input (Function 0AH) 1-45
BUILD BPB 2-4, 2-8, 2-13
Busy bit . . . . 2-9, 2-17 to 2-18

Case mapping 1-108
Change Attributes (Function 43H) . . . 1-120
Change Current Directory
  (Function 3BH) 1-111
Character device 2-1, 2-5
  Example 1-44
B
Check Keyboard Status (Function 0BH) . 1-47
CLOCK device 2-4, 2-19
Close a File Handle (Function 3EH) . . 1-115
Close File (Function 10H) 1-53
Cluster 3-3
Command code field 2-7
Command processor 3-2
COMMAND.COM . . . . 3-1 to 3-2
COMSPEC= 4-3
CON device 2-5
CONFIG.SYS 2-6, 2-12
Console input/output calls 1-3
Control blocks 4-1
Control information 5-1
CONTROL-C Check (Function 33H) 1-102
CONTROL-C Exit Address (Interrupt 23H) 1-19, 3-2
CP/M-compatible calling sequence . . . 1-28

Create a File (Function 3CH) 1-112
Create File (Function 16H) 1-65
Create Sub-Directory (Function 39H) . 1-109
Current Disk (Function 19H) 1-69

DATE 2-19
Delete a Directory Entry (Function 41H) 1-118
Delete File (Function 13H) 1-59
Device drivers 3-7
  Creating 2-5
  Dumb 2-11
  Example 2-20, 2-34
  Installing 2-6
  Smart 2-11
Device header 2-3
Direct Console I/O (Function 06H) . . 1-40
Direct Console Input (Function 07H) . 1-42
Directory entry 1-6
Disk allocation 3-3
Disk Directory 3-4
Disk errors 1-22
Disk format
  IBM 3-3
  MS-DOS 3-7
Disk I/O calls 1-3
Disk Reset (Function 0DH) 1-49
Disk Transfer Address 1-63, 4-3
Display Character (Function 02H) . . . 1-35
Display String (Function 09H) 1-44
Done bit 2-9
Driver 2-2
Dumb device driver 2-11
Duplicate a File Handle (Function 45H) 1-125

Error codes 1-20
Error handling 3-2
Example Block Device Driver 2-20
Example Character Device Driver . . . 2-34
EXE files 5-1
Extended File Control Block 1-6

FAT 1-11, 2-8, 2-13, 3-3, 3-7
FAT ID byte 2-13, 2-15
Fatal Error Abort Address
  (Interrupt 24H) 1-20, 3-2
FCB 4-7
File Allocation Table 1-11, 3-3, 3-7
File Control Block 1-3, 1-51
  Extended 1-6, 4-10
  Fields 1-4, 1-7
  Opened 1-3
  Standard 4-8
  Unopened 1-3
File control Block 4-7
File Size (Function 23H) 1-76
Filename separators 1-88

Filename terminators 1-88
Find Match File (Function 4EH) 1-136
FLUSH 2-18
Flush Buffer (Function 0CH) 1-48
Force Duplicate of Handle
  (Function 46H) 1-126
FORMAT 3-4
Fortran 1-2
Free Allocated Memory (Function 49H) . 1-129
Function call parameters 2-11
Function dispatcher 1-28
Function Request (Interrupt 21H) . . . 1-18, 4-3
Function Requests
  Function 00H 1-33
  Function 01H 1-34
  Function 02H 1-35
  Function 03H 1-36
  Function 04H 1-37
  Function 05H 1-38
  Function 06H 1-40
  Function 07H 1-42
  Function 08H 1-43
  Function 09H 1-44
  Function 0AH 1-45
  Function 0BH 1-47
  Function 0CH 1-48
  Function 0DH 1-49, 1-63
  Function 0EH 1-50
  Function 0FH 1-51, 1-65
  Function 10H 1-53
  Function 11H 1-55
  Function 12H 1-57
  Function 13H 1-59
  Function 14H 1-61
  Function 15H 1-63
  Function 16H 1-65
  Function 17H 1-67
  Function 19H 1-69
  Function 1AH 1-70
  Function 21H 1-72
  Function 22H 1-74
  Function 23H 1-76
  Function 24H 1-78
  Function 25H 1-19, 1-79
  Function 27H 1-81
  Function 28H 1-84
  Function 29H 1-87
  Function 2AH 1-90
  Function 2BH 1-92
  Function 2CH 1-94
  Function 2DH 1-95
  Function 2EH 1-97
  Function 2FH 1-99
  Function 30H 1-100
  Function 31H 1-101

- Function 33H 1-102
- Function 35H 1-104
- Function 36H 1-105
- Function 38H 1-106
- Function 39H 1-109
- Function 3AH 1-110
- Function 3BH 1-111
- Function 3CH 1-112
- Function 3DH 1-113
- Function 3EH 1-115
- Function 3FH 1-116
- Function 40H 1-117
- Function 41H 1-118
- Function 42H 1-119
- Function 43H 1-120
- Function 44H 1-121
- Function 45H 1-125
- Function 46H 1-126
- Function 47H 1-127
- Function 48H 1-128
- Function 49H 1-129
- Function 4AH 1-130
- Function 4BH 1-131
- Function 4CH 1-134
- Function 4DH 1-135
- Function 4EH 1-136
- Function 4FH 1-138
- Function 54H 1-139
- Function 56H 1-140
- Function 57H 1-141

Get Date (Function 2AH) 1-90
Get Disk Free Space (Function 36H) . . 1-105
Get Disk Transfer Address
  (Function 2FH) 1-99
Get DOS Version Number (Function 30H) 1-100
Get Interrupt Vector (Function 35H) . 1-104
Get Time (Function 2CH) 1-94
Get/Set Date /Time of File
  (Function 57H) 1-141

Header 5-1
Hidden files 1-57, 3-5
Hierarchical directories 1-11
High-level languages 1-1

I/O Control for Devices (Function 44H) 1-121, 2-4
IBM disk format 3-3
INIT . . . . 2-5, 2-10 to 2-12
Initial allocation block 1-101
Installable device drivers 2-5
Instruction Pointer 4-4
Internal stack 1-29
Interrupt entry point 2-1
Interrupt handlers 1-19, 4-1
Interrupt-handling routine 1-80

Interrupts 1-14
  Interrupt 20H 1-16, 1-33
  Interrupt 21H 1-18, 1-28
  Interrupt 22H 1-19
  Interrupt 23H . . . . 1-19, 1-34 to 1-35, 1-38, 1-43, 1-45
  Interrupt 24H 1-20
  Interrupt 25H 1-23
  Interrupt 26H 1-25
  Interrupt 27H 1-27
IO.SYS 3-1, 3-6
IOCTL bit 2-4

Keep Process (Function 31H) 1-101

Load and Execute Program
  (Function 4BH) 1-131
Load module . . . . 5-1 to 5-2
Local buffering 2-6
Logical sector 3-7
Logical sector numbers 3-8

Macro 1-10
MEDIA CHECK 2-8, 2-12
Media descriptor byte . . . . 2-10 to 2-11, 2-15
Modify Allocated Memory Blocks
  (Function 4AH) 1-130
Move a Directory Entry (Function 56H) 1-140
Move File Pointer (Function 42H) . . . 1-119
MS-DOS initialization 3-1
MS-DOS memory map 4-1
MS-LINK . . . . 5-1 to 5-2
MSDOS.SYS . . . . 3-1 to 3-2, 3-6
Multiple media 2-11

Name field 2-5
NON DESTRUCTIVE READ NO WAIT 2-17
Non IBM format 2-8
Non IBM format bit 2-4, 2-13
NUL device 2-4

Offset 50H 1-28
Open a File (Function 3DH) 1-113
Open File (Function 0FH) 1-51

Parse File Name (Function 29H) 1-87
Pascal 1-2
PATH 4-3
Pointer to Next Device field 2-3
Print Character (Function 05H) 1-38
Printer input/output calls 1-3
Program segment 4-2
Program Segment Prefix . . . . 1-2 to 1-3, 1-20, 1-28, 4-2
Program Terminate (Interrupt 20H) . . 1-16
PROMPT 4-3

Random Block Read (Function 27H) . . . 1-81
Random Block Write (Function 28H) . . 1-84
Random Read (Function 21H) 1-72
Random Write (Function 22H) 1-74
Read From File/Device (Function 3FH) . 1-116
Read Keyboard (Function 08H) 1-43
Read Keyboard and Echo (Function 01H) 1-34
Read Only Memory 3-1
READ or WRITE 2-16
Record Size 1-63
Registers 1-29
Relocation information 5-1
Relocation item offset value 5-3
Relocation table 5-2
Remove a Directory Entry
  (Function 3AH) 1-110
Rename File (Function 17H) 1-67
Request Header 2-6
Retrieve Return Code (Function 4DH) . 1-135
Return Country-Dependent Info.
  (Function 38H) 1-106
Return Current Setting (Function 54H) 1-139
Return Text of Current Directory
  (Function 47H) 1-127
Returning control to MS-DOS 1-2
ROM 3-1
Root directory 1-11, 3-4

Search for First Entry (Function 11H) 1-55, 4-10
Search for Next Entry (Function 12H) . 1-57
Select Disk (Function 0EH) 1-50
Sequential Read (Function 14H) 1-61
Sequential Write (Function 15H) . . . 1-63
SET 4-3
Set Date (Function 2BH) 1-92
Set Disk Transfer Address
  (Function 1AH) 1-70
Set Relative Record (Function 24H) . . 1-78
Set Time (Function 2DH) 1-95
Set Vector (Function 25H) 1-19, 1-79
Set/Reset Verify Flag (Function 2EH) . 1-97
Smart device driver 2-11
Start segment value 5-3
STATUS 2-18
Status word 2-9
Step Through Directory (Function 4FH) 1-138
Strategy entry point 2-1
Strategy routines 2-5
System files 1-57, 3-5
System prompt 3-2

Terminate a Process (Function 4CH) . . 1-134
Terminate Address (Function 4CH) . . . 4-2
Terminate Address (Interrupt 22H) . . 1-19, 3-2
Terminate But Stay Resident
  (Interrupt 27H) 1-27

Terminate Program (Function 00H) . . . 1-33
TIME 2-19
Type-ahead buffer 2-18

Unit code 2-7
User stack 1-21, 4-1

Volume label 3-5

Wild card characters 1-57, 1-59, 1-88
Write to a File/Device (Function 40H) 1-117

Xenix-compatible calls 1-11

Cut along line

Comment Form

Use this form to comment on this product and to report software problems or errors in the manual. (If you are reporting a software problem, attach a listing if possible.)

Software Description:

Software Product ______________________ Version No. ______

Operating System ______________________ Version No. ______

Other Software Information ______________________

____

Hardware Description:

Seattle Computer Model No. ________ Memory Size __________ KB

Media: Size _____ " Format: Sides _______ Density __________

Other System Hardware ______________________

____

Comments:

Name ______________________ Company ______________

Street ______________________

City ______________________ State ______ Zip ____________

Phone ______________________ Date ______________

Cut along line

Fold and tape

Place
stamp
here

Seattle Computer Products
1114 Industry Drive
Seattle, WA 98188

Attn: Software Support

Fold and tape

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

MASM
Assembler

Microsoft® MS™-DOS

Operating System

Macro Assembler Manual

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft®. MS™-DOS

Operating System

Macro Assembler Manual

Information in this document is subject to change without notice and does not represent a commitment on the part of the Microsoft Corporation. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the *MS-DOS Disk Operating System* on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

Copyright (C) Microsoft Corporation, 1982, 1983

If you have comments about the software or this manual, please complete the Comment Form at the back of the manual and return it to Seattle Computer Products, 1114 Industry Drive, Seattle, WA 98188.

Microsoft is a registered trademark of Microsoft Corporation.

MS is a trademark of Microsoft Corporation.

Intel is at trademark of Intel Corporation.

Part No. 900-001799

System Requirements

Each utility requires different amounts of memory.

Macro Assembler - 96K bytes of memory minimum:
64K bytes for code and static data
32K bytes for run space

Microsoft LINK - 50K bytes of memory minimum:
40K bytes for code
10K bytes for run space

Microsoft LIB - 38K bytes of memory minimum:
28K bytes for code
10K bytes for run space

Microsoft CREF - 24K bytes of memory minimum:
14K bytes for code
10K bytes for run space

Microsoft DEBUG - Memory minimum program-dependent
13K bytes for code
Run space program-dependent

Disk drive(s)

One disk drive if and only if output is sent to the same physical disk from which the input was taken. None of the utility programs allows time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Microsoft LIB Library Manager

Chapter 1 Introduction

Chapter 2 Running MS-LIB

Chapter 3 Error Messages

Index for MS-LIB

Microsoft CREF Cross Reference Utility

Chapter 1 Introduction

Chapter 2 Running MS-CREF

Chapter 3 Error Messages

Chapter 4 Format of MS-CREF Compatible Files

Index for MS-CREF

Microsoft DEBUG Utility

Chapter 1 Introduction

Chapter 2 Commands

Index for DEBUG

General Introduction

Major Features
Using These Manuals
Syntax Notation
Learning More about Assembly Language Programming
Overview of Program Development

Microsoft Macro Assembler Utility

Introduction

Chapter 1 Creating a Macro Assembler Source File

Chapter 2 Names: Labels, Variables, and Symbols

Chapter 3 Expressions: Operands and Operators

Chapter 4 Action: Instructions and Directives

Chapter 5 Assembling a Macro Assembler Source File

Chapter 6 8087 Support

Chapter 7 Macro Assembler Messages

Appendices

Index for Macro Assembler

Microsoft LINK Linker Utility

Chapter 1 Introduction

Chapter 2 MS-LINK Technical Information

Index for MS-LINK

GENERAL INTRODUCTION

The *Microsoft Macro Assembler Manual* includes utility programs used for developing assembly language programs. In addition, the Microsoft LINK Linker Utility and DEBUG are used with of Microsoft's 16-bit language compilers.

Major Features

Macro Assembler Utility

Microsoft's Macro Assembler is a powerful assembler for 8086 based computers.

Macro Assembler supports most of the directives found in Microsoft's Macro Assembler for the 8080. Macros and conditionals are Intel 8080 standard.

Macro Assembler is upward compatible with Intel's ASM-86, except Intel codemacros, macros, and a few $ directives.

Macro Assembler offers relaxed typing so that if you enter a typeless operand for an instruction that accepts only one type of operand, Macro Assembler assembles the statement correctly instead of returning an error message.

Microsoft LINK Linker Utility (Technical Information Only)

MS-LINK is a virtual linker, which can link programs that are larger than available memory.

MS-LINK produces relocatable executable object code.

MS-LINK processes overlays that you define.

MS-LINK can perform multiple library searches, using a dictionary library search method.

MS-LINK prompts you for input and output modules and other link session parameters.

MS-LINK can be run with an automatic response file to answer the Linker prompts.

Microsoft LIB Library Manager

MS-LIB can add, delete, and extract modules in your library of program files.

MS-LIB prompts you for input and output file and module names.

MS-LIB can be run with an automatic response file to answer the library prompts.

MS-LIB produces a cross-reference of symbols in the library modules.

Microsoft CREF Cross-Reference Utility

MS-CREF produces a cross-reference listing of all symbolic names in the Macro Assembler source program, giving both the source line number of the definition and the source line numbers of all other references to the symbols.

Microsoft DEBUG Utility

DEBUG provides a controlled testing environment for binary and executable object files.

DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change.

DEBUG allows you to alter the contents of a file or the contents of a CPU register, and then immediately reexecute a program to check on the validity of the changes.

Using These Manuals

These manuals are designed to be used as a set and individually. Each manual is mostly self-contained and refers to the other manuals only at junctures in the software. The overview given below describes the flow of program development from creating a source file through program execution. The processes described in this overview are echoed and expanded in overviews in each of the manuals contained in the Microsoft Macro Assembler Manual.

Also, note that each manual has its own index.

Figure 1 illustrates an overview of the Microsoft Macro Assembler Manual.

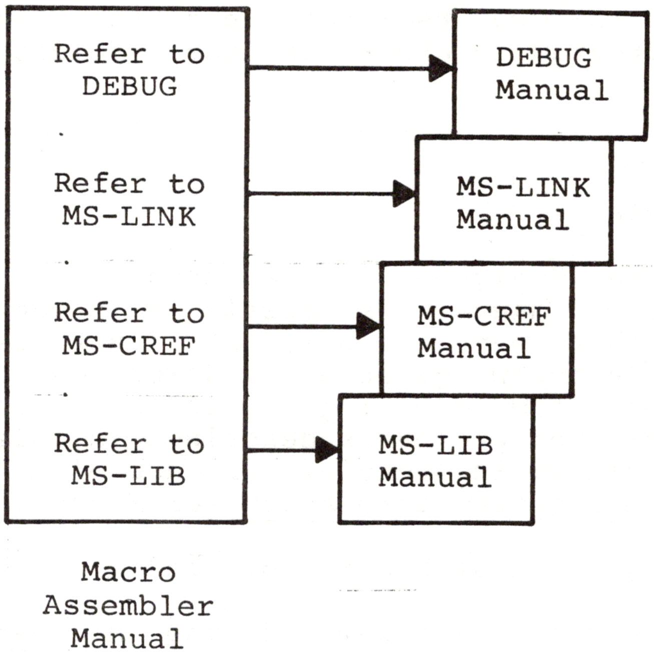

Refer to DEBUG
Refer to MS-LINK
Refer to MS-CREF
Refer to MS-LIB
Macro Assembler Manual
DEBUG Manual
MS-LINK Manual
MS-CREF Manual
MS-LIB Manual

Figure 1. Overview, Macro Assembler Manual

Each of these manuals is used independently. References between manuals reflect junctures in the software.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ]	Square brackets indicate that the enclosed entry is optional.
< >	Angle brackets indicate data you must enter. When the angle brackets enclose lower case text, you must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, you must press the key named by the text; for example, <RETURN>.
{ }	Braces indicate that you have a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.
...	Ellipses indicate that an entry may be repeated as many times as needed or desired.
CAPS	Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

Figure 2 illustrates the syntax notation used in this manual.

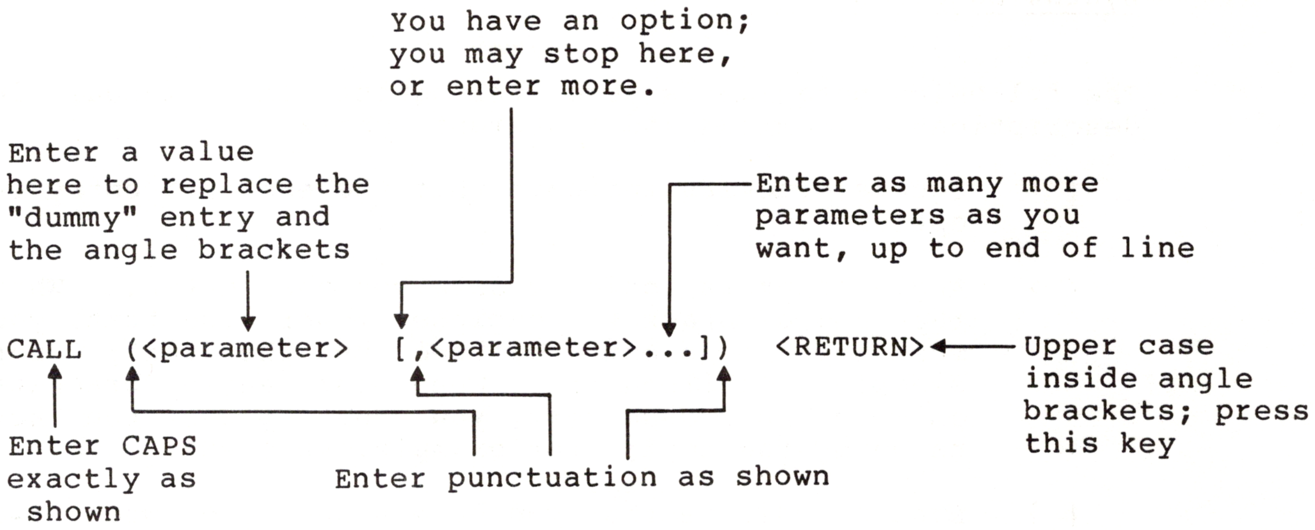

You have an option; you may stop here, or enter more.
Enter a value here to replace the "dummy" entry and the angle brackets
Enter as many more parameters as you want, up to end of line
CALL (<parameter> [,<parameter>...]) <RETURN>
Upper case inside angle brackets; press this key
Enter CAPS exactly as shown
Enter punctuation as shown

Figure 2. Syntax Notation

Learning More about Assembly Language Programming

These manuals explain how to use MS-DOS utilities and features, but they do not teach you how to program in assembly language.

We assume that you have had some experience programming in assembly language. If you do not have any experience, we suggest two courses:

1. Gain some experience on a less sophisticated assembler.

2. Refer to any or all of the following books for assistance:

   Morse, Stephen P. The 8086 Primer. Rochelle Park, NJ: Hayden Publishing Co., 1980.

   Rector, Russell and George Alexy. The 8086 Book. Berkeley, CA: Osbourne/McGraw-Hill, 1980.

   The 8086 Family User's Manual. Santa Clara, CA: Intel Corporation, 1979.

   8086/8087/8088 Macro Assembly Language Reference Manual. Santa Clara, CA: Intel Corporation, 1980.

NOTE

Some of the information in these books was based on preliminary data and may not reflect the final functional state of the microprocessors. Information in your Microsoft manuals was based on Microsoft's development of its 16-bit software for the 8086 and 8088.

Overview of Program Development

This overview describes generally the steps of program development. Each step is described fully in the individual product manuals. The numbers in the descriptions match the numbers in the facing diagram.

1. Use EDLIN (the editor in Microsoft's MS-DOS), or other MS-DOS editor, to create an 8086 assembly language source file. Give the source file the filename extension .ASM (Macro Assembler recognizes .ASM as the default).

2. Assemble the source file with Macro Assembler, which outputs an assembled object file with the default filename extension .OBJ (2a). Assembled files, your program files (2b), can be linked together in step 3.

   Macro Assembler (optionally) creates two types of listing file:

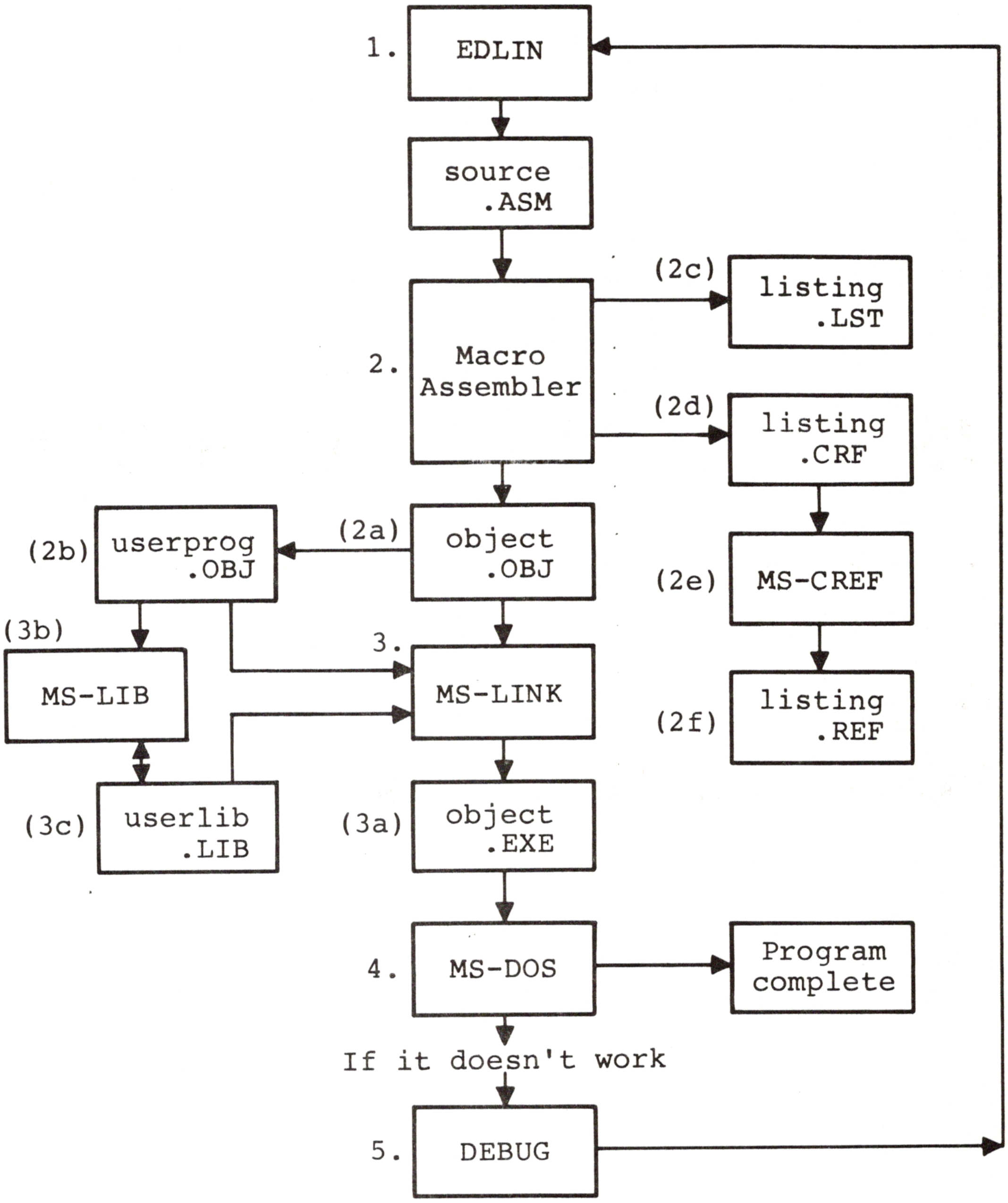
   (2c) a normal listing file which shows assembled code with relative addresses, source statements, and full symbol table;

   (2d) a cross-reference file, a special file with special control characters that allow MS-CREF (2e) to create a list showing the source line number of every symbol's definition and all references to it (2f). When a cross-reference file is created, the normal listing file (with the .LST extension) has line numbers placed into it as references for line numbers following symbols in the cross-reference listing.

3. Link one or more .OBJ modules together, using MS-LINK, to produce an executable object file with the default filename extension .EXE (3a).

   While developing your program, you may want to create a library file for MS-LINK to search to resolve external references. Use MS-LIB (3b) to create user library file(s) (3c) from existing library files (3c) and/or user program object files (2b).

4. Run your assembled and linked program, the .EXE file (3a), under MS-DOS (4). If your program does not run properly, use the DEBUG utility to locate any errors.


1.
EDLIN
source .ASM
2.
Macro Assembler
(2c)
listing .LST
(2d)
listing .CRF
(2e)
MS-CREF
(2f)
listing .REF
(2a)
object .OBJ
(2b)
userprog .OBJ
(3b)
MS-LIB
3.
MS-LINK
(3c)
userlib .LIB
(3a)
object .EXE
4.
MS-DOS
Program complete
If it doesn't work
5.
DEBUG

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® Macro Assembler

Utility

for 8086 and 8088 Microprocessors

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® Macro Assembler

Utility

for 8086 and 8088 Microprocessors

System Requirements

The Macro Assembler Utility requires 96K bytes of memory minimum:

64K bytes for code and static data
32K bytes for run space

Disk drive(s)

One disk drive if and only if output is sent to the same physical disk from which the input was taken. The Macro Assembler Utility does not allow time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

Introduction

Features of Macro Assembler 1

Overview of Macro Assembler Operation 8

Chapter 1 Creating a Macro Assembler Source File

1.1 General Facts about Source Files 1-1

1.2 Statement Line Format 1-5

1.3 Names 1-6

1.4 Comments 1-8

1.5 Action 1-9

1.6 Expressions 1-10

Chapter 2 Names: Labels, Variables, and Symbols

2.1 Labels 2-2

2.2 Variables 2-5

2.3 Symbols 2-7

Chapter 3 Expressions: Operands and Operators

3.1 Memory Organization 3-2

3.2 Operands 3-8

3.3 Operators 3-17

Chapter 4 Action: Instructions and Directives

4.1 Instructions 4-2

4.2 Directives 4-3

Chapter 5 Assembling a Macro Assembler Source File

5.1 How to Start Macro Assembler 5-1

5.2 Macro Assembler Command Characters 5-4

5.3 Macro Assembler Command Prompts 5-5

5.4 Macro Assembler Command Switches 5-7

5.5 Formats of Listings and Symbol Tables 5-10

Chapter 6 8087 Support

6.1 Switches 6-1

Chapter 7 Macro Assembler Messages

7.1 Operating Messages 7-1

7.2 Error Messages 7-2

Appendix A ASCII Character Codes

Appendix B Table of Macro Assembler Directives

Appendix C Table of 8086 and 8087 Instructions

Index

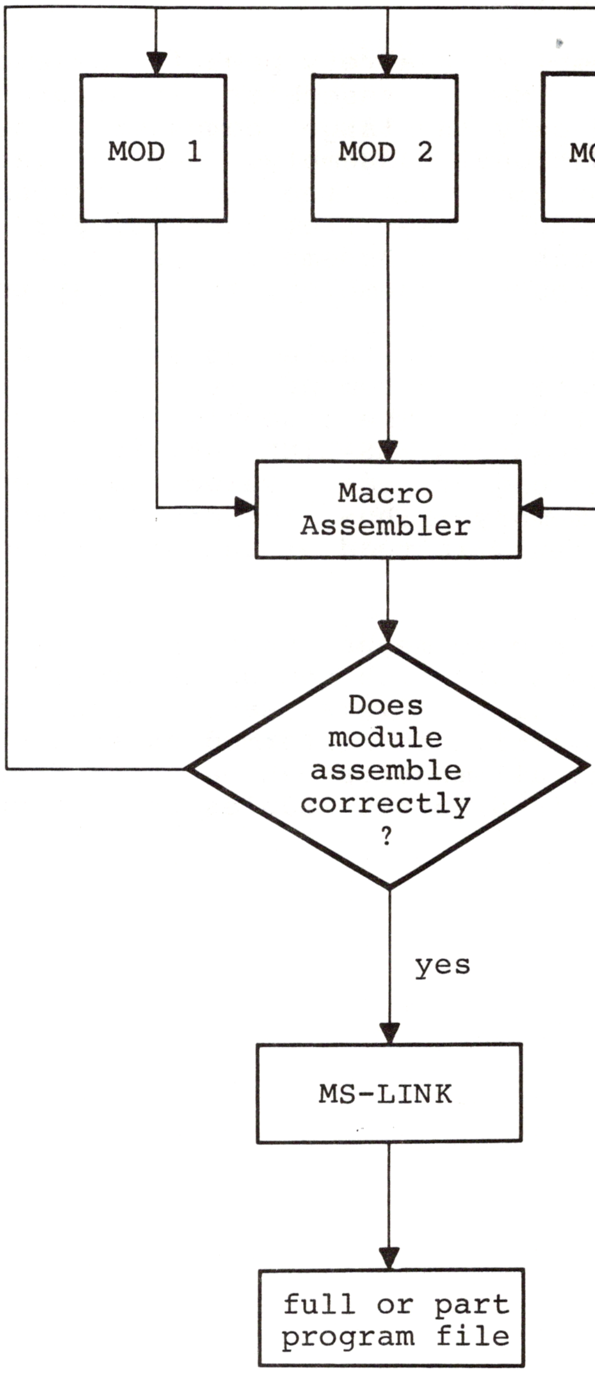
INTRODUCTION

Features of Macro Assembler

Microsoft's Macro Assembler is a very powerful assembler for 8086-based computers. Macro Assembler incorporates many features usually found only in large computer assemblers. Macro assembly, conditional assembly, and a variety of assembler directives provide all the tools necessary to derive full use and full power from an 8086, 8087 or 8088 microprocessor. Although Macro Assembler is more complex than any other microcomputer assembler, it is easy to use.

Macro Assembler produces relocatable object code. Each instruction and directive statement is given a relative offset from its segment base. The assembled code can then be linked using Microsoft's MS-LINK utility to produce relocatable, executable object code. Relocatable code can be loaded anywhere in memory. Thus, the program can execute where it is most efficient, instead of in some fixed range of memory addresses.

In addition, relocatable code means that programs can be created in modules, each of which can be assembled, tested, and perfected individually. This saves recoding time because testing and assembly are performed on smaller pieces of program code. Also, all modules can be error-free before being linked together into larger modules or into the whole program.


MOD 1
MOD 2
MOD 3
Macro Assembler
Does module assemble correctly ?
yes
MS-LINK
full or part program file

Individual modules can be edited and assembled until they work correctly.

When the individual modules are ready, they can be linked singly or into one or more larger modules.

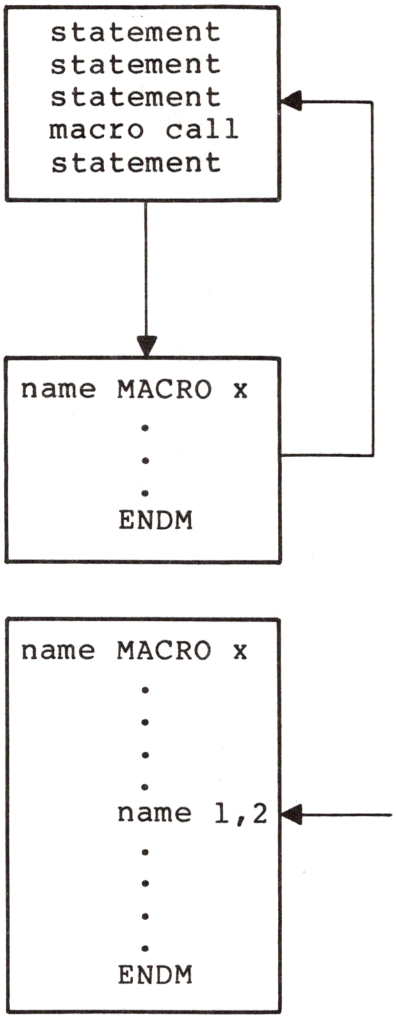
Figure 1. The Assembly Process

Macro Assembler supports Microsoft's complete 8080 macro facility, which is Intel 8080 standard. The macro facility permits the writing of blocks of code for a set of instructions used frequently. The need for recoding these instructions each time they are required in the program is eliminated.

These blocks of code are called macros. The instructions are the macro definition. Each time the set of instructions is needed, instead of recoding the set of instructions, a simple "call" to a macro is placed in the source file. Macro Assembler expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to the assembler for use during macro expansion. The use of macros reduces the size of a source module because the macro definitions are given only once; other occurrences are one-line calls.

Macros can be "nested," that is, a macro can be called from inside another macro block. Nesting of macros is limited only by memory.

The macro facility includes repeat, indefinite repeat, and indefinite repeat character directives for programming repeat block operations. The MACRO directive can also be used to alter the action of any instruction or directive by using its name as the macro name. When any instruction or directive statement is placed in the program, Macro Assembler first checks the symbol table it created to see if the instruction or directive is a macro name. If it is, Macro Assembler "expands" the macro call statement by replacing it with the body of instructions in the macro's definition. If the name is not defined as a macro, Macro Assembler tries to match the name with an instruction or directive. The MACRO directive also supports local symbols and conditional exiting from the block if further expansion is unnecessary.


statement
statement
statement
macro call
statement
name MACRO x
ENDM
name MACRO x
name 1,2
ENDM

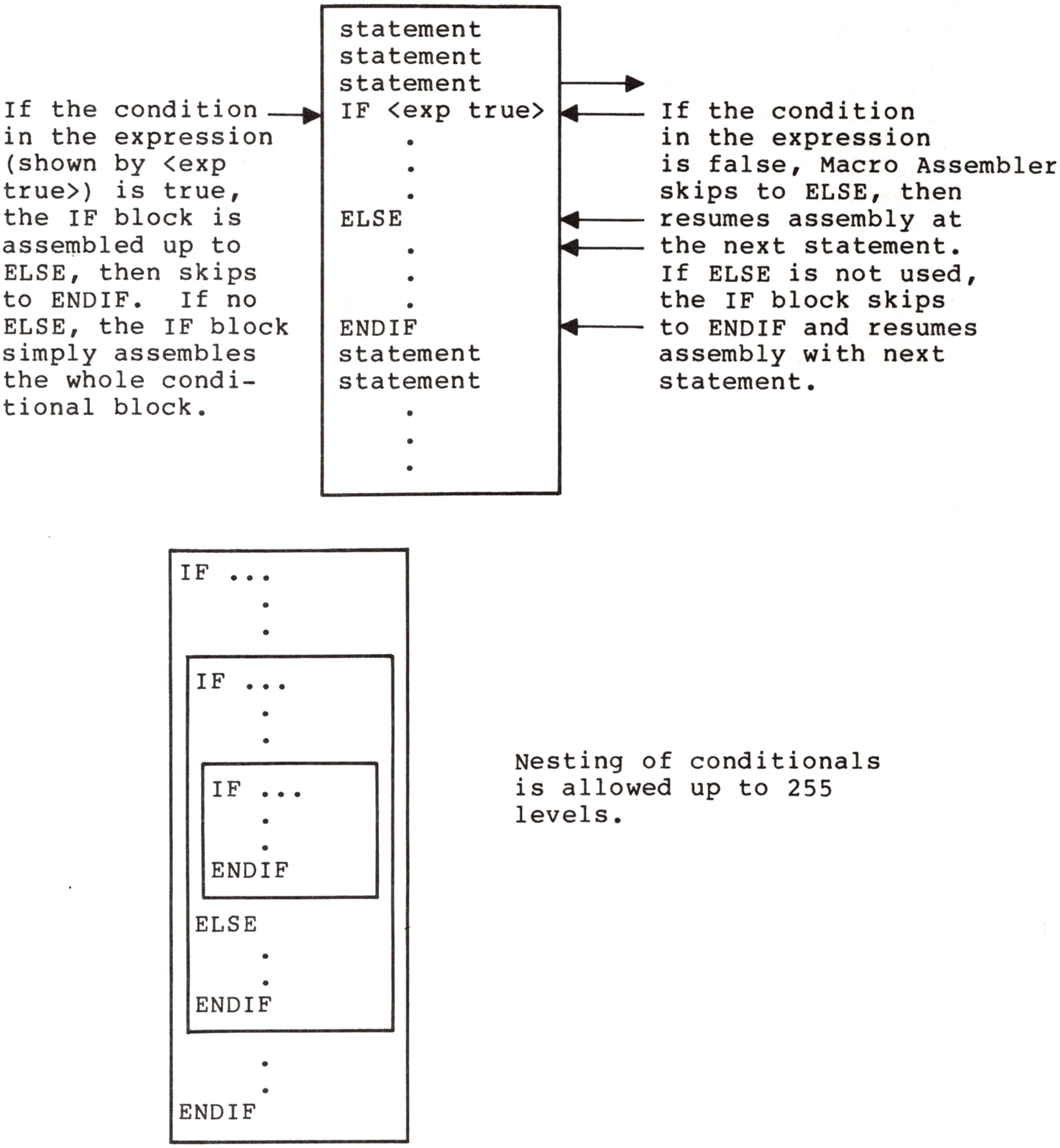
When the assembler encounters a macro call, it finds the MACRO block and replaces the call with the block of statements that define the macro.

Nested MACRO call: name defined elsewhere as a macro, is "expanded" during assembly, as shown above.

Figure 2. Assembler Macros

Macro Assembler supports an expanded set of conditional directives. Directives for evaluating a variety of assembly conditions can test assembly results and branch where required. Unneeded or unwanted portions of code will be left unassembled. Macro Assembler can test for blank or nonblank arguments, for defined or undefined symbols, for equivalence, for first assembly pass or second, and can compare strings for identity or difference. The conditional directives simplify the evaluation of assembly results, and make programming the testing code for conditions easier.

Macro Assembler's conditional assembly facility also supports conditionals inside conditionals ("nesting"). Conditional assembly blocks can be nested up to 255 levels.



```
statement
statement
statement
IF <exp true>
     .
     .
     .
ELSE
     .
     .
     .
ENDIF
statement
statement
     .
     .
     .
```

If the condition in the expression (shown by <exp true>) is true, the IF block is assembled up to ELSE, then skips to ENDIF. If no ELSE, the IF block simply assembles the whole conditional block.

If the condition in the expression is false, Macro Assembler skips to ELSE, then resumes assembly at the next statement. If ELSE is not used, the IF block skips to ENDIF and resumes assembly with next statement.

```
IF ...
     .
     .
  IF ...
     .
     .
    IF ...
     .
     .
    ENDIF
  ELSE
     .
     .
  ENDIF
     .
     .
ENDIF
```

Nesting of conditionals is allowed up to 255 levels.

Figure 3. Conditional Statements

Macro Assembler supports all the major 8080 directives found in Microsoft's Macro Assembler for the 8080 processor. This means that any conditional, macro, or repeat blocks programmed under the 8080 Macro Assembler can be used under Macro Assembler for the 8086. Processor instructions and some directives (e.g., .PHASE, CSEG, DSEG) within the blocks will need to be converted to the 8086 instruction set. All the major Macro Assembler directives (pseudo-ops) for the 8080 that are supported under Macro Assembler for the 8086 will assemble as is, as long as the expressions to the directives are correct for the processor and the program. The syntax of directives is unchanged. Macro Assembler is upwardly-compatible, Macro Assembler for the 8080 processor and with Intel's ASM86(R), except Intel codemacros and macros.

Some 8086 instructions take only one operand type. If a typeless operand is entered for an instruction that accepts only one type of operand (e.g., in the instruction PUSH [BX], [BX] has no size, but PUSH only takes a word), it would be wasteful to return an error for a lapse of memory or a typographical error. When the wrong type choice is given, Macro Assembler displays an error message but generates the "correct" code. That is, it always outputs instructions, not just NOP instructions. For example, if you enter:

```
                          MOV   AL,WORDLBL
                               /    |   \
You may have                  /     |    \(2)
meant one of                 /      |     MOV AL,BYTE PTR WORDLBL
three instructions:         /       |(3)
                           /     MOV AL,<other>
                       (1)/
                          MOV AX,WORDLBL
```

Macro Assembler generates instruction (2) because it assumes that when you specify a register, you mean that register and that size; therefore, the other operand is the "wrong size." Macro Assembler accordingly modifies the "wrong" operand to fit the register size (in this case) or the size of whatever is the most likely "correct" operand in an expression. This eliminates some mundane debugging chores. An error message is still returned, however, because you may have misstated the operand the Macro Assembler assumes is "correct."

Overview of Macro Assembler Operation

The first task in developing a program is to create a source file. Use EDLIN (the resident editor in Microsoft's MS-DOS operating system), or any other 8086 editor compatible with your operating system, to create the Macro Assembler source file. Macro Assembler assumes a default filename extension of .ASM for the source file. Creating the source file involves creating instruction and directive statements that follow the rules and constraints described in Chapters 1-4 in this manual.

When the source file is ready, run Macro Assembler as described in Chapter 5, "Assembling a Macro Assembler Source File." Refer to Chapter 7, "Macro Assembler Messages," for explanations of any messages displayed during or immediately after assembly.

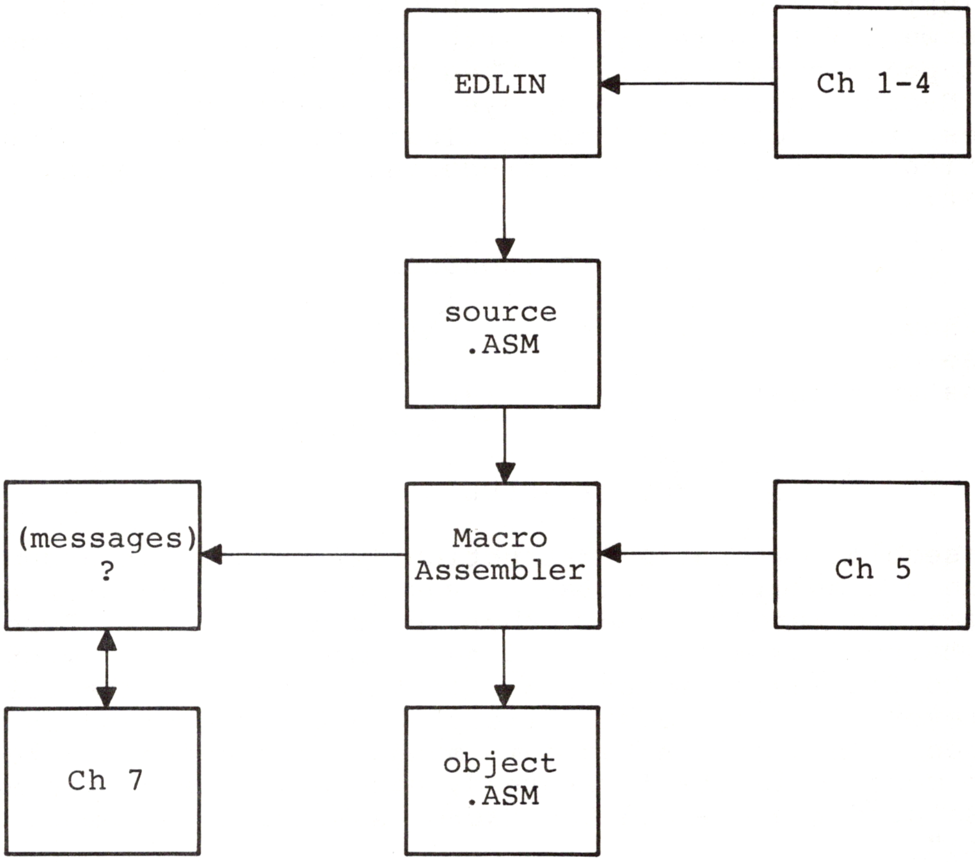


Figure 4. Overview of Macro Assembler Operation

Macro Assembler is a two-pass assembler. This means that the source file is assembled twice. But slightly different actions occur during each pass. During the first pass, the assembler:

> evaluates the statements and expands macro call statements
>
> calculates the amount of code it will generate
>
> builds a symbol table where all symbols, variables, labels, and macros are assigned values

During the second pass, the assembler

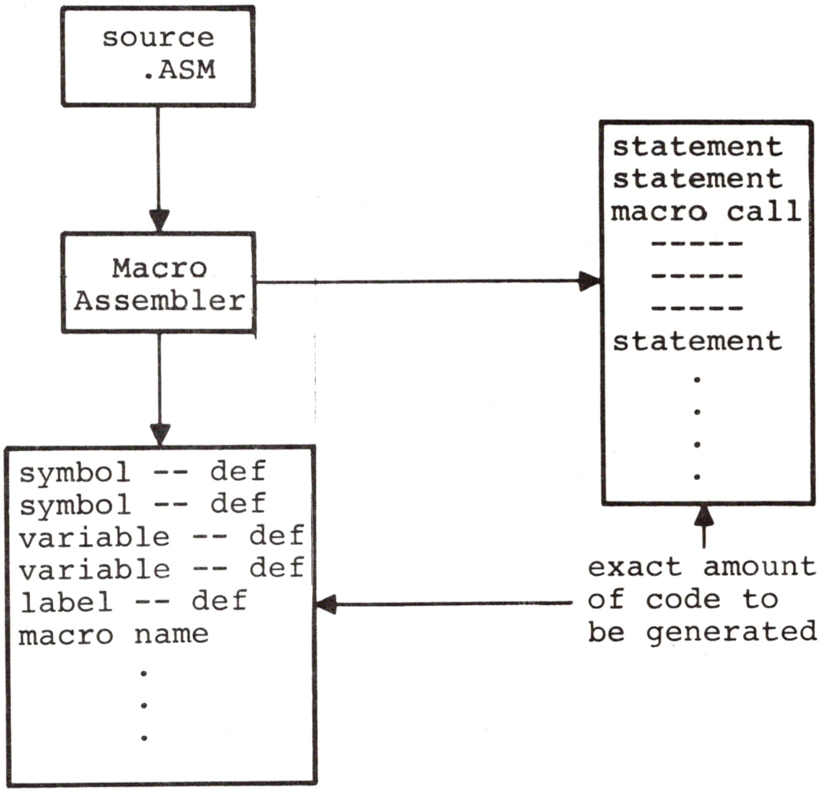
> fills in the symbol, variable, label, and expression values from the symbol table
>
> expands macro call statements
>
> emits the relocatable object code into a file with the default filename extension .OBJ

The .OBJ file is suitable for processing with the Microsoft LINK utility (MS-LINK). The .OBJ file can be stored as part of the user's library of object programs, which later can be linked with one or more .OBJ modules by MS-LINK (refer to the MS-LINK utility for further explanation and instructions). The .OBJ modules can also be processed with the Microsoft LIB Library Manager (refer to the *Microsoft LIB Library Manager Manual* for further explanation and instructions).

The source file can also be assembled without creating an .OBJ file. All the other assembly steps are performed, but the object code is not sent to disk. Only erroneous source statements are displayed on the terminal screen. This practice is useful for checking the source code for errors.

It is faster than creating an .OBJ file because no file is created or written. Modules can be test assembled quickly and errors corrected before the object code is put on disk. Modules that assemble without errors do not clutter the disk.

PASS 1


source
.ASM
Macro
Assembler
statement
statement
macro call

-----
statement
.
.
.
.
symbol -- def
symbol -- def
variable -- def
variable -- def
label -- def
macro name
.
.
.
exact amount
of code to
be generated

PASS 2

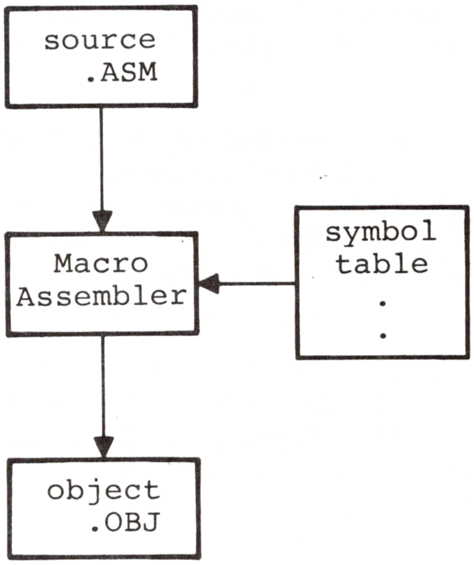

source
.ASM
Macro
Assembler
symbol
table
.
.
object
.OBJ

Figure 5. Pass 1 and Pass 2

Macro Assembler will create, on command, a listing file and a cross-reference file. The listing file contains the beginning relative addresses (offsets from segment base) assigned to each instruction, the machine code translation of each statement (in hexadecimal values), and the statement itself. The listing also contains a symbol table which shows the values of all symbols, labels, and variables, plus the names of all macros. The listing file receives the default filename extension .LST.

The cross-reference file contains a compact representation of variables, labels, and symbols. The cross-reference file receives the default filename extension .CRF. When this cross-reference file is processed by Microsoft CREF (MS-CREF), the file is converted into an expanded symbol table that lists all the variables, labels, and symbols in alphabetical order; followed by the line number in the source program where each is defined; followed by the line numbers where each is used in the program. The final cross-reference listing receives filename extension .REF. (Refer to the Microsoft CREF Cross-Reference Utility Manual for further explanation and instructions.)

Figure 6 illustrates the files that Macro Assembler can produce.

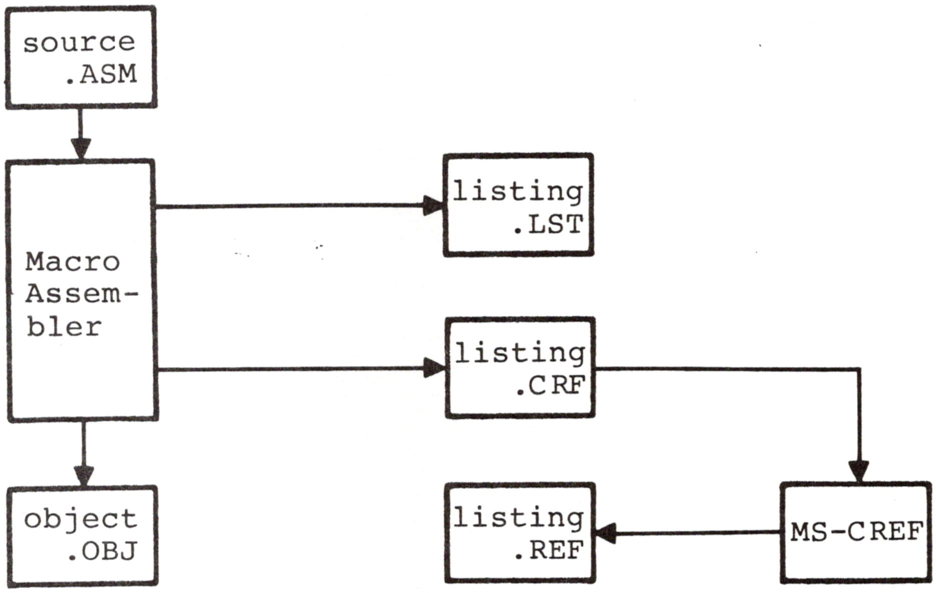


Figure 6. Files That Macro Assembler Produces

Contents

Chapter 1 Creating a Macro Assembler Source File

1.1 General Facts about Source Files 1-1
  Naming Your Source File 1-1
  Legal Characters 1-2
  Numeric Notation 1-3
  What's in a Source File? 1-4
1.2 Statement Line Format 1-5
1.3 Names 1-6
1.4 Comments 1-8
1.5 Action 1-9
1.6 Expressions 1-10

CHAPTER 1

CREATING A MACRO ASSEMBLER SOURCE FILE

To create a source file for Macro Assembler, you need to use an editor program, such as EDLIN in Microsoft's MS-DOS. You simply create a program file as you would for any other assembly or high-level programming language. Use the general facts and specific descriptions in this chapter and the three following chapters when creating the file.

This chapter discusses the statement format and introduces descriptions of its components. In Chapter 2, you will find full descriptions of names: variables, labels, and symbols. Chapter 3 provides full descriptions of expressions and their components, operands and operators. Chapter 4 includes full descriptions of the assembler directives.

1.1 GENERAL FACTS ABOUT SOURCE FILES

Naming Your Source File

When you create a source file, you must name it. A filename may be any name that is legal for your operating system. When you run Macro Assembler to assemble your source file, Macro Assembler assumes that your source filename has the extension .ASM.

You do not need to give your source filename the .ASM extension. However, if your source filename has has an extension other than .ASM, you must specify the extension name when you run Macro Assembler. (You do not need to specify the .ASM extension if your source filename has an extension of .ASM. Macro Assembler will supply the default extension for you.)

Note that Macro Assembler gives the object file it outputs the default extension .OBJ. To avoid confusion or the destruction of your source file, you should avoid giving a source file an extension of .OBJ. For similar reasons, you should also avoid the extensions .EXE, .LST, .CRF, and .REF.

Legal Characters

The legal characters for your symbol names are:

A-Z 0-9 ? @ _ $

Only the numerals (0-9) cannot appear as the first character of a name (a numeral must appear as the first character of a numeric value).

Additional special characters act as operators or delimiters:

:	(colon)--segment override operator
.	(period)--operator for field name of Record or Structure; may be used in a filename only if it is the first character
[ ]	(square brackets--around register names to indicate value in address in register, not value (data) in register
( )	(parentheses)--operator in DUP expressions and operator to change precedence of operator evaluation
< >	(angle brackets) operators used around initialization values for Records or Structure, around parameters in IRP macro blocks, and to indicate literals

The square brackets and angle brackets are also used for syntax notation in the discussions of the assembler directives (Section 4.2, "Directives"). When these characters are operators and not syntax notation, you are told explicitly; for example, "angle brackets must be coded as shown."

Numeric Notation

The default input radix for all numeric values is decimal. The output radix for all listings is hexadecimal for code and data items and decimal for line numbers. The output radix can only be changed to octal radix by giving the /O switch when Macro Assembler is run (see Section 5.4, "Macro Assembler Command Switches"). There are two ways to change the input radix:

1. With the .RADIX directive (see Section 4.2.1, "Memory Directives")
2. By special notation appended to a numeric value:

Radix	Range	Notation	Example
Binary	0-1	B	01110100B
Octal	0-7	Q or O	735Q or 621O
Decimal	0-9	none or D	9384 (default) 8149D*
Hexadecimal	0-9 A-F	H	OFFH or 80H**

* When .RADIX directive changes default radix to not decimal.
**First character must be numeral from 0-9.

What's in a Source File?

A source file for Macro Assembler consists of instruction statements and directive statements. Instruction statements are made of 8086 instruction mnemonics and their operands, which command specific processes directly to the 8086 processor. Directive statements are commands to Macro Assembler to prepare data for use in and by instructions.

Statement line format is described in Section 1.2. The parts of a statement are described in Sections 1.3-1.6 and in Chapters 2-4. Statements are usually placed in blocks of code assigned to a specific segment (code, data, stack, extra). The segments may appear in any order in the source file. Within the segments, generally speaking, statements may appear in any order that creates a valid program. Some exceptions to random ordering do exist, which will be discussed under the affected assembler directives.

Every segment must end with an end segment statement (ENDS); every procedure must end with an end procedure statement (ENDP); and every structure must end with an end structure statement (ENDS). Likewise, the source file must end with an END statement that tells Macro Assembler where program execution should begin.

Section 3.1, "Memory Organization," describes how segments, groups, the ASSUME directive, and the SEG operator relate to one another and to your programming as a whole. This information is important and helpful for developing your programs. The information is presented in Chapter 3 as a prelude to the discussion of operands and operators.

1.2 STATEMENT LINE FORMAT

Statements in source files follow a strict format, which allows some variation.

Macro Assembler directive statements consist of four "fields": Name, Action, Expression, Comment. For example:

```
FOO     DB      0D5E            ;create variable FOO
                                ;containing the value 0D5EH

 |       |        |
Name    Action  Expression      ;Comment
```

Macro Assembler instruction statements usually consist of three "fields": Action, Expression, Comment. For example:

```
        MOV     CX,FOO          ;here's the count number
         |        |                |
        Action  Expression      ;Comment
```

An instruction statement may have a Name field under certain circumstances; see the discussion in Section 1.3, "Names."

1.3 NAMES

The name field, when present, is the first entry on the statement line. The name may begin in any column, although normally names are started in column 1.

Names may be any length you choose. However, Macro Assembler considers only the first 31 characters significant when your source file is assembled.

One other significant use for names is with the MACRO directive. Although all the rules covering names, described in Chapter 2, apply to MACRO names, the discussion of macro names is better left to the section describing the macro facility.

Macro Assembler supports the use of names in a statement line for three purposes: to represent code, to represent data, and to represent constants.

To make a name represent code, use:

NAME: followed by a directive, instruction, or nothing at all
NAME LABEL NEAR (for use inside its own segment only)

NAME LABEL FAR (for use outside its own segment)

EXTRN NAME:NEAR (for use outside its own module but inside its own segment only)

EXTRN NAME:FAR (for use outside its own module and segment)

To make a name represent data, use:

NAME LABEL <size> (BYTE, WORD, etc.)

NAME Dx <exp>

EXTRN NAME:<size> (BYTE, WORD, etc.)

To make a name represent a constant, use:

```
NAME EQU <constant>

NAME = <constant>

NAME SEGMENT <attributes>

NAME GROUP <segment-names>
```

1.4 COMMENTS

Comments are never required for the successful operation of an assembly language program, but they are strongly recommended.

If you use comments in your program, every comment on every line must be preceded by a semicolon. If you want to place a very long comment in your program, you can use the COMMENT directive. The COMMENT directive releases you from the required semicolon on every line (refer to COMMENT in Section 4.2.1, "Memory Directives").

Comments document the processing that is supposed to happen at a particular point in a program. When comments are used in this manner, they can be useful for debugging, for altering code, or for updating code. Consider putting comments at the beginning of each segment, procedure, structure, module, and after each line in the code that begins a step in the processing.

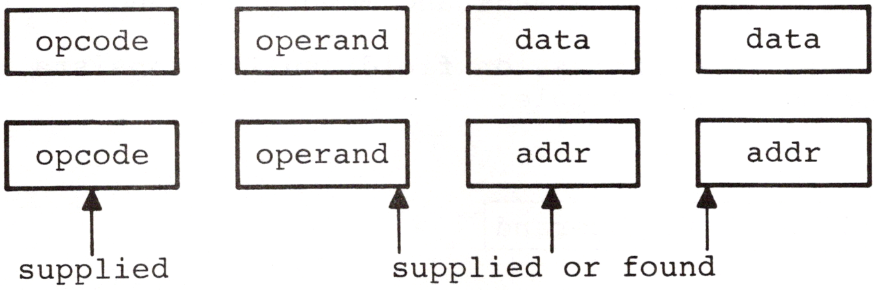
Comments are ignored by Macro Assembler. Comments do not add to the memory required to assemble or to run your program, except in macro blocks where comments are stored with the code.

1.5 ACTION

The action field contains either an 8086 instruction mnemonic or a Macro Assembler assembler directive. Refer to Section 4.1, "Instructions," for a general discussion and to Appendix C for a list of 8086 instruction mnemonics. The Macro Assembler directives are described in detail in Section 4.2, "Directives."

If the name field is blank, the action field will be the first entry in the statement format. In this case, the action may appear in any column, 1 through maximum line length (minus columns for action and expression).

The entry in the action field either directs the processor to perform a specific function or it directs the assembler to perform one of its functions. Instructions tell the processor to perform some action. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expression in instruction statements

(opcode is the action part of an instruction)

Directives give the assembler directions for I/O, memory organization, conditional assembly, listing and cross-reference control, and definitions.

1.6 EXPRESSIONS

The expression field contains entries which are operands and/or combinations of operands and operators.

Some instructions take no operands; some take one, and others take two. For two-operand instructions, the expression field consists of a destination operand and a source operand, in that order, separated by a comma. For example:

opcode dest-operand , source-operand

For one-operand instructions, the operand is a source or a destination operand, depending on the instruction. If one or both of the operands is omitted, the instruction carries that information in its internal coding.

Source operands are immediate operands, register operands, memory operands, or attribute operands. Destination operands are register operands and memory operands.

For directives, the expression field usually consists of a single operand. For example:

directive operand

A directive operand is a data operand, a code (addressing) operand, or a constant, depending on the nature of the directive.

For many instructions and directives, operands may be connected with operators to form a longer operand that looks like a mathematical expression. These operands are called complex operands. Use of a complex operand permits you to specify addresses or data derived from several places. For example:

```
MOV   FOO[BX],AL
```

The destination operand is the result of adding the address represented by the variable FOO and the address found in register BX. The processor is instructed to move the value in register AL to the destination calculated from these two operand elements. Another example:

```
        MOV   AX,FOO+5[BX]
```

In this case, the source operand is the result of adding the value represented by the symbol FOO plus 5 plus the value found in the BX register.

Macro Assembler supports the following operands and operators in the expression field (shown in order of precedence):

Operands

- Immediate (incl. symbols)
- Register
- Memory
  - label
  - variables
    - simple
    - indexed
    - structures
- Attribute
  - override
    - PTR
    - :(seg)
    - SHORT
    - HIGH
    - LOW
  - value returning
    - OFFSET
    - SEG
    - THIS
    - TYPE
    - .TYPE
    - LENGTH
    - SIZE
  - record specifying
    - FIELD
    - MASK
    - WIDTH

Operators

LENGTH, SIZE, WIDTH, MASK, FIELD [ ], ( ), < >

segment override(:)

PTR, OFFSET, SEG, TYPE, THIS

HIGH, LOW

*, /, MOD, SHL, SHR

+, -(unary), -(binary)

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

SHORT, .TYPE

NOTE

Some operators can be used as operands or as part of an operand expression. Refer to Sections 3.2, "Operands," and 3.3, "Operators," for details of operands and operators.

Contents

Chapter 2 Names: Labels, Variables, and Symbols

2.1 Labels 2-2
2.2 Variables 2-5
2.3 Symbols 2-7

CHAPTER 2

NAMES: LABELS, VARIABLES, AND SYMBOLS

Names are used in several ways throughout Macro Assembler, wherever any naming is allowed or required.

Names are symbolic representations of values. The values may be addresses, data, or constants.

Names may be any length you choose. However, Macro Assembler will truncate names longer than 31 characters when your source file is assembled.

Names may be defined and used in a number of ways. This chapter introduces you to the basic ways to define and use names. You will discover additional uses as you study the chapters on Expressions and Action, and as you use Macro Assembler.

Macro Assembler supports three types of names in statement lines: labels, variables, and symbols. This chapter covers how to define and use these three types of names.

2.1 LABELS

Labels are names used as targets for JMP, CALL, and LOOP instructions. Macro Assembler assigns an address to each label as it is defined. When you use a label as an operand for JMP, CALL, or LOOP, Macro Assembler can substitute the attributes of the label for the label name, sending processing to the appropriate place.

Labels are defined in one of four ways:

1. <name>:

   Use a name followed immediately by a colon. This defines the name as a NEAR label. <name>: may be prefixed to any instruction and to all directives that allow a Name field. <name>: may also be placed on a line by itself.

   Examples:

   ```
   CLEAR_SCREEN:   MOV   AL,20H
   FOO:   DB   0FH
   SUBROUTINE3:
   ```

2. <name>   LABEL   NEAR
   <name>   LABEL   FAR

   Use the LABEL directive. Refer to the discussion of the LABEL directive in Section 4.2.1, "Memory Directives."

   NEAR and FAR are discussed under the Type Attribute below.

   Examples:

   ```
   FOO   LABEL   NEAR
   GOO   LABEL   FAR
   ```

3. <name>   PROC   NEAR
   <name>   PROC   FAR

   Use the PROC directive. Refer to the discussion of the PROC directive in Section 4.2.1, "Memory Directives."

   NEAR is optional because it is the default if you enter only <name> PROC. NEAR and FAR are discussed under the Type Attribute below.

Examples:

```
REPEAT    PROC    NEAR
CHECKING    PROC    ;same as CHECKING PROC NEAR
FIND_CHR    PROC    FAR
```

4. EXTRN <name>:NEAR
   EXTRN <name>:FAR

   Use the EXTRN directive.

   NEAR and FAR are discussed under the Type Attribute below.

   Refer to the discussion of the EXTRN directive in Section 4.2.1, "Memory Directives."

   Examples:

```
EXTRN FOO:NEAR
EXTRN ZOO:FAR
```

A label has four attributes: segment, offset, type, and the CS ASSUME in effect when the label is defined. Segment is the segment where the label is defined. Offset is the distance from the beginning of the segment to the label's location. Type is either NEAR or FAR.

Segment

Labels are defined inside segments. The segment must be assigned to the CS segment register to be addressable. The segment may be assigned to a group, in which case the group must be addressable through CS. Macro Assembler requires that a label be addressable through the CS register. Therefore, the segment (or group) attribute of a symbol is the base address of the segment (or group) where it is defined.

Offset

The offset attribute is the number of bytes from the beginning of the label's segment to where the label is defined. The offset is a 16-bit unsigned number.

Type

Labels are one of two types: NEAR or FAR. NEAR labels are used for references from within the segment where the label is defined. NEAR labels may be referenced from more than one module, as long as the references are from a segment with the same name and attributes and have the same CS ASSUME.

FAR labels are used for references from segments with a different CS ASSUME, or when there are more than 64K bytes between the label reference and the label definition.

NEAR and FAR cause Macro Assembler to generate slightly different code. NEAR labels supply their offset attribute only (a 2-byte pointer). FAR labels supply both their segment and offset attributes (a 4-byte pointer).

2.2 VARIABLES

Variables are names used in expressions as operands to instructions and directives. A variable represents an address where a specified value may be found.

Variables look much like labels and are defined alike in some ways. The differences are important.

Variables are defined three ways:

1.
```
<name> <define-dir>          ;no colon!
<name> <struc-name> <expression>
<name> <rec-name> <expression>
```

   <define-dir> is any of the five Define directives: DB, DW, DD, DQ, DT

   Example:

```
    START_MOVE   DW   ?
```

   <struc-name> is a structure name defined by the STRUC directive.

   <rec-name> is a record name defined by the RECORD directive.

   Examples:

```
    CORRAL   STRUC
               .
               .
               .
             ENDS
    HORSE    CORRAL  <'SADDLE'>
```

   Note that HORSE will have the same size as the structure CORRAL.

```
    GARAGE   RECORD  CAR:8='P'

    SMALL    GARAGE  10 DUP(<'Z'>)
```

   Note that SMALL will have the same size as the record GARAGE.

   See the DEFINE, STRUC, and RECORD directives in Section 4.2.1, "Memory Directives."

2.
```
<name> LABEL <size>
```

   Use the LABEL directive with one of the size specifiers.

<size> is one of the following size specifiers:

BYTE - specifies 1 byte
WORD - specifies 2 bytes
DWORD - specifies 4 bytes
QWORD - specifies 8 bytes
TBYTE - specifies 10 bytes

Example:

```
CURSOR   LABEL   WORD
```

See LABEL directive in Section 4.2.1, "Memory Directives."

3. EXTRN <name>:<size>

Use the EXTRN directive with one of the size specifiers described above. See EXTRN directive in Section 4.2.1, "Memory Directives."

Example:

```
EXTRN FOO:DWORD
```

Variables also have the three attributes segment, offset, and type (as do labels).

Segment and Offset are the same for variables as for labels. The Type attribute is different.

Type

The type attribute is the size of the variable's location, as specified when the variable is defined. The size depends on which Define directive was used or which size specifier was used to define the variable.

Directive	Type	Size
DB	BYTE	1 byte
DW	WORD	2 bytes
DD	WORD	4 bytes
DQ	QWORD	8 bytes
DT	TBYTE	10 bytes

2.3 SYMBOLS

Symbols are names defined without reference to a Define directive or to code. Like variables, symbols are also used in expressions as operands to instructions and directives.

Symbols are defined three ways:

1. <name> EQU <expression>

   Use the EQU directive. See EQU directive in Section 4.2.1, "Memory Directives."

   <expression> may be another symbol, an instruction mnemonic, a valid expression, or any other entry (such as text or indexed references).

   Examples:

   ```
       FOO    EQU    7H
       ZOO    EQU    FOO
   ```

2. <name> = <expression>

   Use the equal sign directive. See Equal Sign directive in Section 4.2.1, "Memory Directives."

   <expression> may be any valid expression.

   Examples:

   ```
       GOO    =    0FH
       GOO    =    $+2
       GOO    =    GOO+FOO
   ```

3. EXTRN <name>:ABS

   Use the EXTRN directive with type ABS. See EXTRN directive in Section 4.2.1, "Memory Directives."

   Example:

   ```
       EXTRN BAZ:ABS
   ```

   BAZ must be defined by an EQU or = directive to a valid expression.

Contents

Chapter 3 Expressions: Operands and Operators

3.1 Memory Organization 3-2
3.2 Operands 3-8
3.2.1 Immediate Operands 3-9
    Data Items 3-9
    Symbols 3-9
3.2.2 Register Operands 3-10
3.2.3 Memory Operands 3-13
    Direct Memory Operands 3-13
    Indexed Memory Operands 3-14
    Structure Operands 3-15
3.3 Operators 3-17
3.3.1 Attribute Operators 3-17
    Override Operators 3-18
    Value Returning Operators 3-23
    Record Specific Operators 3-29
3.3.2 Arithmetic Operators 3-33
3.3.3 Relational Operators 3-34
3.3.4 Logical Operators 3-35
3.3.5 Expression Evaluation 3-36
    Precedence of Operators 3-36

CHAPTER 3

EXPRESSIONS: OPERANDS AND OPERATORS

Chapter 1 provided a brief introduction to expressions. Basically, expression is the term used to indicate values on which an instruction or directive performs its functions.

Every expression consists of at least one operand (a value). An expression may consist of two or more operands. Multiple operands are joined by operators. The result is a series of elements that looks like a mathematical expression.

This chapter describes the types of operands and operators that Macro Assembler supports. The discussion of memory organization in a Macro Assembler program acts as a preface to the descriptions of operands and operators, and as a link to topics discussed in Chapter 2.

3.1 MEMORY ORGANIZATION

Most of your assembly language program is written in segments. In the source file, a segment is a block of code that begins with a SEGMENT directive statement and ends with an ENDS directive. In an assembled and linked file, a segment is any block of code that is addressed through the same segment register and is not more than 64K bytes long.

You should note that Macro Assembler leaves everything relating to segments to MS-LINK. MS-LINK resolves all references. For that reason, Macro Assembler does not check (because it cannot) to see if your references are entered with the correct distance type. Values such as OFFSET are also left to MS-LINK to resolve.

Although a segment may not be more than 64K bytes long, you may, as long as you observe the 64K limit, divide a segment among two or more modules. (The SEGMENT statement in each module must be the same.)

When the modules are linked together, the several segments become one. References to labels, variables, and symbols within each module acquire the offset from the beginning of the whole segment, not just from the beginning of their portion of the whole segment. (All divisions are removed.)

You have the option of grouping several segments into a group using the GROUP directive. When you group segments, you tell Macro Assembler that you want to be able to refer to all of these segments as a single entity. (This does not eliminate segment identity, nor does it make values within a particular segment less immediately accessible. It does make value relative to a group base.) The advantage of grouping is that you can refer to data items without worrying about segment overrides or changing segment registers.

With this in mind, you should note that references within segments or groups are relative to a segment register. Thus, until linking is completed, the final offset of a reference is relocatable. For this reason, the OFFSET operator does not return a constant. The major purpose of OFFSET is to cause Macro Assembler to generate an immediate instruction; that is, to use the address of the value instead of the value itself.

There are two kinds of references in a program:

1. Code references - JMP, CALL, LOOPxx - These references are relative to the address in the CS register. (You cannot override this assignment.)

2. Data references - all other references - These references are usually relative to the DS register, but this assignment may be overridden.

When you give a forward reference in a program statement, for example:

```
MOV AX,<ref>
```

Macro Assembler first looks for the segment of the reference. Macro Assembler scans the segment registers for the SEGMENT of the reference, then the GROUP (if any) of the reference.

However, the use of the OFFSET operator always returns the offset relative to the segment. If you want the offset relative to a GROUP, you must override this restriction by using the GROUP name and the colon operator. For example:

```
MOV AX,OFFSET <group-name>:<ref>
```

If you set a segment register to a group with the ASSUME directive, then you may also override the restriction on OFFSET by using the register name. For example:

```
MOV AX,OFFSET DS:<ref>
```

The result of both of these statements is the same.

Code labels have four attributes:

1. Segment - what segment the label belongs to

2. Offset - the number of bytes from the beginning of its segment

3. Type - NEAR or FAR

4. CS ASSUME - the CS ASSUME the label was coded under

When you enter a NEAR JMP or NEAR CALL, you are changing the offset (IP) in CS. Macro Assembler compares the CS ASSUME of the target (where the label is defined) with the current CS ASSUME. If they are different, Macro Assembler returns an error (you must use a FAR JMP or FAR CALL).

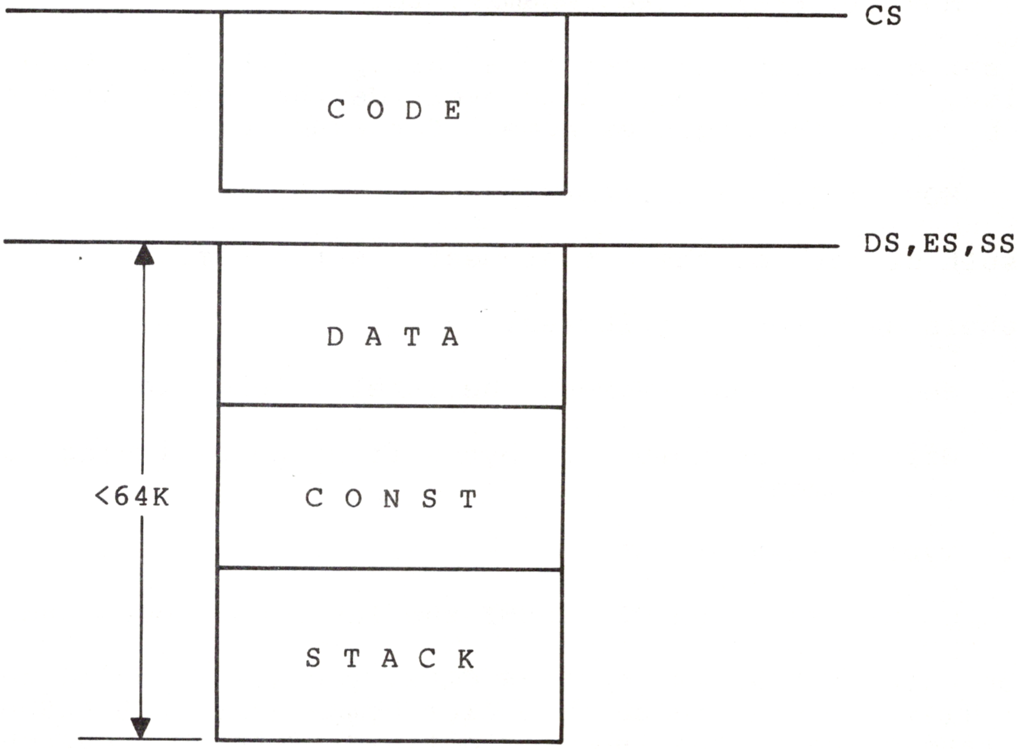
When you enter a FAR JMP or FAR CALL, you are changing both the offset (IP) in CS and the paragraph number. The paragraph number is changed to the CS ASSUME of the target address.

Let's take a common case, a segment called CODE, and a group (called DGROUP) that contains three segments (called DATA, CONST, and STACK).

The program statements would be:

```
DGROUP  GROUP   DATA,CONST,STACK
        ASSUME  CS:CODE,DS:DGROUP,SS:DGROUP,ES:DGROUP
        MOV     AX,DGROUP          ;CS initialized by entry;
        MOV     DS,AX              ;you initialize DS, do this
                                   ;as soon as possible,
                                   ;especially before any
                                   ;DS relative references
          .
          .
          .
```

As a diagram, this arrangement could be represented as follows:



Given this arrangement, a statement like

```
MOV AX,<variable>
```

causes Macro Assembler to find the best segment register to reach this variable. (The "best" register is the one that requires no segment overrides.)

A statement like

```
MOV AX,OFFSET <variable>
```

tells Macro Assembler to return the offset of the variable relative to the beginning of the variable's segment.

If this <variable> is in the CONST segment and you want to reference its offset from the beginning of DGROUP, you need a statement like the following:

```
MOV AX,OFFSET DGROUP:<variable>
```

Macro Assembler is a two-pass assembler. During pass 1, it builds a symbol table and calculates how much code is generated, but does not produce object code. If undefined items are found (including forward references), assumptions are made about the reference so that the correct number of bytes are generated on pass 1. Only certain types of errors are displayed: errors involving items that must be defined on pass 1. No listing is produced unless a /D switch is given when you run the assembler. The /D switch produces a listing for both passes.

On pass 2, the assembler uses the values defined in pass 1 to generate the object code. Definitions of references during pass 2 are checked against the pass 1 value, which is in the symbol table. Also, the amount of code generated during pass 1 must match the amount generated during pass 2. If either is different, Macro Assembler returns a phase error.

Because pass 1 must keep correct track of the relative offset, some references must be known on pass 1. If they are not known, the relative offset will not be correct.

The following references must be known on pass 1:

1. IF/IFE <expression>
   If <expression> is not known on pass 1, Macro Assembler does not know to assemble the conditional block (or which part to assemble if ELSE is used). On pass 2, the assembler would know and would assemble, resulting in a phase error.

2. <expression> DUP(...)
   This operand explicitly changes the relative offset, so <expression> must be known on pass 1. The value in parentheses need not be known because it does not affect the number of bytes generated.

3. .RADIX <expression>
   Because this directive changes the input radix, constants could have a different value, which could cause Macro Assembler to evaluate IF or DUP statements incorrectly.

The biggest problem for the assembler is handling forward references. How can it know the kind of a reference when it still has not seen the definition? This is one of the main reasons for two passes. And, unless Macro Assembler can tell from the statement containing the forward reference what the size, the distance, or any other of its attributes are, the assembler can only take the safe route (generate the largest possible instruction in some cases, except for segment override or FAR). This results in extra code that does nothing. (Macro Assembler figures this out by pass 2, but it cannot reduce the size of the instructions without causing an error, so it puts out NOP instructions (90H).

For this reason, Macro Assembler includes a number of operators to help the assembler. These operators tell Macro Assembler what size instruction to generate when it is faced with an ambiguous choice. As a benefit, you can also reduce the size of your program by using these operators to change the nature of the arguments to the instructions.

Examples:

```
MOV AX,FOO ;FOO = forward constant
```

This statement causes Macro Assembler to generate a move from memory instruction on pass 1. By using the OFFSET operator, we can cause Macro Assembler to generate an immediate operand instruction.

```
MOV AX,OFFSET FOO  ;OFFSET says use the address
                   ;of FOO
```

Because OFFSET tells Macro Assembler to use the address of FOO, the assembler knows that the value is immediate. This method saves a byte of code.

Similarly, if you have a CALL statement that calls to a label that may be in a different CS ASSUME, you can prevent problems by attaching the PTR operator to the label:

```
CALL FAR PTR <forward-label>
```

At the opposite extreme, you may have a JMP forward that is less than 127 bytes. You can save yourself a byte if you use the SHORT operator.

```
JMP SHORT <forward-label>
```

However, you must be sure that the target is indeed within 127 bytes or Macro Assembler will not find it.

The PTR operator can be used another way to save yourself a byte when using forward references. If you defined FOO as a forward constant, you might enter the statement:

```
MOV [BX],FOO
```

You may want to refer to FOO as a byte immediate. In this case, you could enter either of these statements (they are equivalent):

```
MOV BYTE PTR [BX],FOO

MOV [BX],BYTE PTR FOO
```

These statements tell Macro Assembler that FOO is a byte immediate. A smaller instruction is generated.

3.2 OPERANDS

An operand may be any one of three types: Immediate, Register, or Memory operands. There is no restriction on combining the types of operands.

The following list shows all the types and the items that comprise them:

- Immediate operands
    - Data items
    - Symbols
- Register operands
- Memory operands
    - Direct
        - Labels
        - Variables
        - Offset (fieldname)
    - Indexed
        - Base register
        - Index register
        - [constant]
        - +displacement
    - Structure

3.2.1 Immediate Operands

Immediate operands are constant values that you supply when you type a statement line. The value may be typed either as a data item or as a symbol.

Instructions that take two operands permit an immediate operand as the source operand only (the second operand in an instruction statement). For example:

```
MOV AX,9
```

Data Items

Macro Assembler recognizes values in forms other than decimal when special notation is appended. The default input radix is decimal. Any numeric values entered without numeric notation appended will be treated as a decimal value. These other values include ASCII characters as well as numeric values.

Data Form	Format	Example
Binary	xxxxxxxxB	01110001B
Octal	xxxO xxxQ	735O (letter O) 412Q
Decimal	xxxxx xxxxxD	65535 (default) 1000D (when .RADIX changes input radix to nondecimal)
Hexadecimal	xxxxH	0FFFFH (1st digit must be 0-9)
ASCII	'xx' "xx"	'OM' (more than two with DB only; "OM" both forms are synonymous)
10 real	xx.xxE&+xx	25.23E-7 (floating point format)
16 real	x...xR	8F76DEA9R (1st digit must be 0-9; the total number of digits must be 8, 16, or 20; or 9, 17, 21 if first digit is 0)

Symbols

Symbol names equated with some form of constant information (see Section 2.3, "Symbols") may be used as immediate operands. Using a symbol constant in a statement is the same as using a numeric constant. Therefore, using the sample statement above, you could type:

```
        MOV AX,FOO
```

assuming FOO was defined as a constant symbol. For example:

```
        FOO EQU 9
```

3.2.2 Register Operands

The 8086 processor contains a number of registers. These registers are identified by two-letter symbols that the processor recognizes (the symbols are reserved).

The registers are appropriated to different tasks: general registers, pointer registers, counter registers, index registers, segment registers, and a flag register.

The general registers are two sizes: 8-bit and 16-bit. All other registers are 16-bit.

The general registers are both 8-bit and 16-bit registers. Actually, the 16-bit general registers are composed of a pair of 8-bit registers, one for the low byte (bits 0-7) and one for the high byte (bits 8-15). Note, however, that each 8-bit general register can be used independently from its mate. In this case, each 8-bit register contains bits 0-7.

Segment registers are initialized by the user and contain segment base values. The segment register names (CS, DS, SS, ES) can be used with the colon segment override operator to inform Macro Assembler that an operand is in a different segment than specified in an ASSUME statement. (See the segment override operator in Section 3.3.1, "Attribute Operators.)"

The flag register is one 16-bit register containing nine 1-bit flags (six arithmetic flags and three control flags).

Each of the registers (except segment registers and flags) can be an operand in arithmetic and logical operations.

Register/Memory Field Encoding:

MOD=11		
R/M	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register Mode

EFFECTIVE ADDRESS CALCULATION			
R/M	MOD=00	MOD=01	MOD=10
000	[BX]+[SI]	[BX]+[SI]+D8	[BX]+[SI]+D16
001	[BX]+[DI]	[BX]+[DI]+D8	[BX]+[DI]+D16
010	[BP]+[SI]	[BP]+[SI]+D8	[BP]+[SI]+D16
011	[BP]+[DI]	[BP]+[DI]+D8	[BP]+[DI]+D16
100	[SI]	[SI]+D8	[SI]+D16
101	[DI]	[DI]+D8	[DI]+D16
110	DIRECT ADDRESS	[BP]+D8	[BP]+D16
111	[BX]	[BX]+D8	[BX]+D16

Note: D8 = a byte value; D16 = a word value

Other Registers:

Segment:	CS	code segment
	DS	data segment
	SS	stack segment
	ES	extra segment

Flags:

1-bit arithmetic flags		3 1-bit control flags	
CF	carry flag	DF	direction flag
PF	parity flag	IF	interrupt-enable flag
AF	auxiliary flag	TF	trap flag
ZF	zero flag		
SF	sign flag		

NOTE

The BX, BP, SI, and DI registers are also used as memory operands. The distinction is: when these registers are enclosed in square brackets [ ], they are memory operands; when they are not enclosed in square brackets, they are register operands (see Section 3.2.3, "Memory Operands").

3.2.3 Memory Operands

A memory operand represents an address in memory. When you use a memory operand, you direct Macro Assembler to an address to find some data or instruction.

A memory operand always consists of an offset from a base address.

Memory operands fit into three categories: those that do not use a register (direct memory operands), those that use a base or index register (indexed memory operands), and structure operands.

Direct Memory Operands

Direct memory operands do not use a register, and consist of a single offset value. Direct memory operands are labels, simple variables, and offsets.

Memory operands can be used as destination operands as well as source operands for instructions that take two operands. For example:

```
MOV AX,FOO
MOV FOO,CX
```

Indexed Memory Operands

Indexed memory operands use base and index registers, constants, displacement values, and variables, often in combination. When you combine indexed operands, you create an address expression.

Indexed memory operands use square brackets to indicate indexing (by a register or by registers) or subscripting (for example, FOO[5]). The square brackets are treated like plus signs (+). Therefore,

```
FOO[5] is equivalent to FOO+5
5[FOO] is equivalent to 5+FOO
```

The only difference between square brackets and plus signs occurs when a register name appears inside the square brackets. Then, the operand is indexed.

The types of indexed memory operands are:

Base registers: [BX] [BP]

BP has SS as its default segment register;
all others have DS as default.

Index registers: [DI] [SI]

[constant] Immediate in square brackets [8], [FOO]

+Displacement 8-bit or 16-bit value.
Used only with another indexed operand.

These elements may be combined in any order. The only restriction is that two base registers and two indexed registers cannot be combined:

```
[BX+BP] ;illegal
[SI+DI] ;illegal
```

Some examples of indexed memory operand combinations:

```
[BP+8]
[SI+BX][4]
16[DI+BP+3]
8[FOO]-8
```

More examples of equivalent forms:

```
5[BX][SI]
BX+5][SI]
[BX+SI+5]
[BX]5[SI]
```

Structure Operands

Structure operands take the form <variable>.<field>.

<variable> is any name you give when coding a statement line that initializes a Structure field. The <variable> may be an anonymous variable, such as an indexed memory operand.

<field> is a name defined by a DEFINE directive within a STRUC block. <field> is a typed constant.

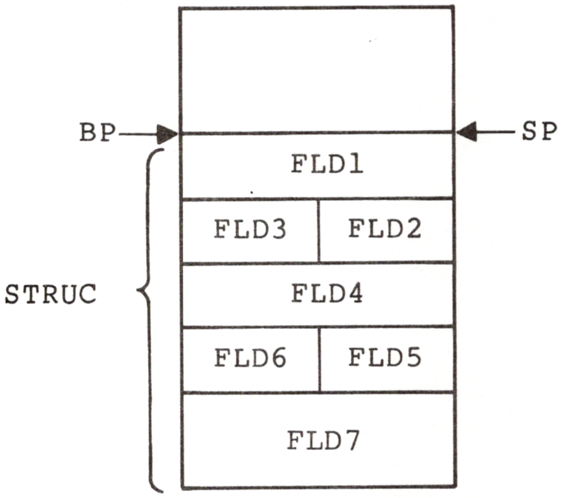
The period (.) must be included.

Example:

```
ZOO      STRUC
GIRAFFE  DB  ?
ZOO      ENDS

LONG_NECK  ZOO <16>

MOV AL,LONG_NECK.GIRAFFE

MOV AL,[BX].GIRAFFE   ,anonymous variable
```

The use of structure operands can be helpful in stack operations. If you set up the stack segment as a structure, setting BP to the top of the stack (BP equal to SP), then you can access any value in the stack structure by field name indexed through BP; for example:

```
[BP].FLD6
```



This method makes all values on the stack available all the time, not just the value at the top. Therefore, this method makes the stack a handy place to pass parameters to subroutines.

3.3 OPERATORS

An operator may be one of four types: attribute, arithmetic, relational, or logical.

Attribute operators are used with operands to override their attributes, return the value of the attributes, or to isolate fields of records.

Arithmetic, relational, and logical operators are used to combine or compare operands.

3.3.1 Attribute Operators

Attribute operators used as operands perform one of three functions:

Override an operand's attributes

Return the values of operand attributes

Isolate record fields (record specific operators)

The following list shows all the attribute operators by type:

- Override operators
  - PTR
  - colon (:) (segment override)
  - SHORT
  - THIS
  - HIGH
  - LOW
- Value returning operators
  - SEG
  - OFFSET
  - TYPE
  - .TYPE
  - LENGTH
  - SIZE
- Record specific operators
  - Shift count (Field name)
  - WIDTH
  - MASK

Override Operators

These operators are used to override the segment, offset, type, or distance of variables and labels.

Pointer (PTR)

<attribute> PTR <expression>

The PTR operator overrides the type (BYTE, WORD, DWORD) or the distance (NEAR, FAR) of an operand.

<attribute> is the new attribute; the new type or new distance.

<expression> is the operand whose attribute is to be overridden.

The most important and frequent use for PTR is to assure that Macro Assembler understands what attribute the expression is supposed to have. This is especially true for the type attribute. Whenever you place forward references in your program, PTR will make clear the distance or type of the expression. This way you can avoid phase errors.

The second use of PTR is to access data by type other than the type in the variable definition. Most often this occurs in structures. If the structure is defined as WORD but you want to access an item as a byte, PTR is the operator for this. However, a much easier method is to enter a second statement that defines the structure in bytes, too. This eliminates the need to use PTR for every reference to the structure. Refer to the LABEL directive in Section 4.2.1, "Memory Directives."

Examples:

```
CALL WORD PTR [BX][SI]
MOV BYTE PTR ARRAY

ADD BYTE PTR FOO,9
```

Segment Override (:) (colon)

```
<segment-register>:<address-expression>
<segment-name>:<address-expression>
<group-name>:<address-expression>
```

The segment override operator overrides the assumed segment of an address expression (which may be a label, a variable, or other memory operand).

The colon operator helps with forward references by telling the assembler to what a reference is relative (segment, group, or segment register).

Macro Assembler assumes that labels are addressable through the current CS register. Macro Assembler also assumes that variables are addressable through the current DS register, or possibly the ES register, by default. If the operand is in another segment and you have not alerted Macro Assembler through the ASSUME directive, you will need to use a segment override operator. Also, if you want to use a nondefault relative base (that is, not the default segment register), you will need to use the segment override operator for forward references. Note that if Macro Assembler can reach an operand through a nondefault segment register, it will use it, but the reference cannot be forward in this case.

<segment-register> is one of the four segment register names: CS, DS, SS, ES.

<segment-name> is a name defined by the SEGMENT directive.

<group-name> is a name defined by the GROUP directive.

Examples:

```
MOV AX,ES:[BX+SI]

MOV CSEG:FAR_LABEL,AX

MOV AX,OFFSET DGROUP:VARIABLE
```

SHORT

SHORT <label>

SHORT overrides NEAR distance attributes of labels used as targets for the JMP instruction. SHORT tells Macro Assembler that the distance between the JMP statement and the <label> specified as its operand is not more than 127 bytes either direction.

The major advantage of using the SHORT operator is to save a byte. Normally, the <label> carries a 2-byte pointer to its offset in its segment. Because a range of 256 bytes can be handled in a single byte, the SHORT operator eliminates the need for the extra byte (which would carry 00 or FF anyway). However, you must be sure that the target is within +127 bytes of the JMP instruction before using SHORT.

Example:

```
                JMP SHORT REPEAT
                  .
                  .
                  .
      REPEAT:
```

THIS

THIS <distance>
THIS <type>

The THIS operator creates an operand. The value of the operand depends on which argument you give THIS.

The argument to THIS may be:

1. A distance (NEAR or FAR)

2. A type (BYTE, WORD, or DWORD)

THIS <distance> creates an operand with the distance attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

THIS <type> creates an operand with the type attribute you specify, an offset equal to the current location counter, and the segment attribute (segment base address) of the enclosing segment.

Examples:

```
TAG EQU THIS BYTE same as TAG LABEL BYTE

SPOT_CHECK = THIS NEAR same as
SPOT_CHECK LABEL NEAR
```

HIGH,LOW

```
HIGH <expression>
LOW <expression>
```

HIGH and LOW are provided for 8080 assembly language compatibility. HIGH and LOW are byte isolation operators.

HIGH isolates the high 8 bits of an absolute 16-bit value or address expression.

LOW isolates the low 8 bits of an absolute 16-bit value or address expression.

Examples:

```
MOV AH,HIGH WORD_VALUE ;get byte with sign bit

MOV AL,LOW 0FFFFH
```

Value Returning Operators

These operators return the attribute values of the operands that follow them but do not override the attributes.

The value returning operators take labels and variables as their arguments.

Because variables in Macro Assembler have three attributes, you need to use value returning operators to isolate single attributes, as follows:

SEG	isolates the segment base address
OFFSET	isolates the offset value
TYPE	isolates either type or distance
LENGTH and SIZE	isolate the memory allocation

SEG

SEG <label>
SEG <variable>

SEG returns the segment value (segment base address) of the segment enclosing the label or variable.

Example:

```
MOV AX,SEG VARIABLE_NAME
MOV AX,<segment-variable>:<variable>
```

OFFSET

OFFSET <label>
OFFSET <variable>

OFFSET returns the offset value of the variable or label within its segment (the number of bytes between the segment base address and the address where the label or variable is defined).

OFFSET is chiefly used to tell the assembler that the operand is an immediate operand.

NOTES

> OFFSET does <u>not</u> make the value a constant. Only MS-LINK can resolve the final value.
>
> OFFSET is not required with uses of the DW or DD directives. The assembler applies an implicit OFFSET to variables in address expressions following DW and DD.

Example:

```
MOV BX,OFFSET FOO
```

If you use an ASSUME to GROUP, OFFSET will not automatically return the offset of a variable from the base address of the group. Rather, OFFSET will return the segment offset, unless you use the segment override operator (group-name version). If the variable GOB is defined in a segment placed in DGROUP, and you want the offset of GOB in the group, you need to enter a statement like:

```
MOV BX,OFFSET DGROUP:GOB
```

You must be sure that the GROUP directive precedes any reference to a group name, including its use with OFFSET.

TYPE

TYPE <label>
TYPE <variable>

If the operand is a variable, the TYPE operator returns a value equal to the number of bytes of the variable type, as follows:

BYTE = 1
WORD = 2
DWORD = 4
QWORD = 8
TBYTE = 10
STRUC = the number of bytes declared by STRUC

If the operand is a label, the TYPE operator returns NEAR (FFFFH) or FAR (FFFEH).

Examples:

```
MOV AX,(TYPE FOO_BAR) PTR [BX+SI]
```

.TYPE

.TYPE <variable>

The .TYPE operator returns a byte that describes two characteristics of the <variable>: 1) the mode, and 2) whether it is External or not. The argument to .TYPE may be any expression (string, numeric, logical). If the expression is invalid, .TYPE returns zero.

The byte that is returned is configured as follows:

The lower two bits are the mode. If the lower two bits are:

0	the mode is Absolute
1	the mode is Program Related
2	the mode is Data Related

The high bit (80H) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is not External.

The Defined bit is 20H. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external. If neither bit is on, the expression is invalid.

.TYPE is usually used inside macros, where an argument type may need to be tested to make a decision regarding program flow; for example, when conditional assembly is involved.

Example:

```
FOO     MACRO    X
        LOCAL    Z
Z       =    .TYPE X
IF      Z...
```

.TYPE tests the mode and type of X. Depending on the evaluation of X, the block of code beginning with IF Z... may be assembled or omitted.

LENGTH

LENGTH <variable>

LENGTH accepts only one variable as its argument.

LENGTH returns the number of type units (BYTE, WORD, DWORD, QWORD, TBYTE) allocated for that variable.

If the variable is defined by a DUP expression, LENGTH returns the number of type units duplicated; that is, the number that precedes the first DUP in the expression.

If the variable is not defined by a DUP expression, LENGTH returns 1.

Examples:

```
FOO DW 100 DUP(1)

MOV CX,LENGTH FOO ;get number of elements
                  ;in array
                  ;LENGTH returns 100

BAZ DW 100 DUP(1,10 DUP(?))
```

LENGTH BAZ is still 100, regardless of the expression following DUP.

```
GOO DD (?)
```

LENGTH GOO returns 1 because only one unit is involved.

SIZE

SIZE <variable>

SIZE returns the total number of bytes allocated for a variable.

SIZE is the product of the value of LENGTH times the value of TYPE.

Example:

```
FOO DW 100 DUP(1)

MOV BX,SIZE FOO  ;get total bytes in array

SIZE = LENGTH X TYPE
SIZE =   100  X WORD
SIZE =   100  X 2
SIZE = 200
```

Record Specific Operators

Record specific operators are used to isolate fields in a record.

Records are defined by the RECORD directive (see Section 4.2.1, "Memory Directives"). A record may be up to 16 bits long. The record is defined by fields, which may be from one to 16 bits long. To isolate one of the three characteristics of a record field, you use one of the record specific operators, as follows:

Shift count	Number of bits from low end of record to low end of field (number of bits to right shift the record to lowest bits of record)
WIDTH	The number of bits wide the field or record is (number of bits the field or record contains)
MASK	Value of record if field contains its maximum value and all other fields are zero (all bits in field contain 1; all other bits contain 0)

In the following discussions of the record specific operators, the following symbols are used:

FOO	a record defined by the RECORD directive `FOO RECORD FIELD1:3,FIELD2:6,FIELD3:7`
BAZ	a variable used to allocate FOO `BAZ FOO < >`

FIELD1, FIELD2, and FIELD3 are the fields of the record FOO.

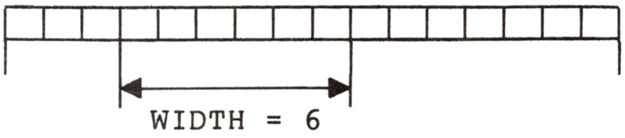
Shift-count - (record-fieldname)

<record-fieldname>

The shift count is derived from the record fieldname to be isolated.

The shift count is the number of bits the field must be shifted right to place the lowest bit of the field in the lowest bit of the record byte or word.

If a 16-bit record (FOO) contains three fields (FIELD1, FIELD2, and FIELD3), the record can be diagrammed as follows:



FIELD1 has a shift count of 13.
FIELD2 has a shift count of 7.
FIELD3 has a shift count of 0.

When you want to isolate the value in one of these fields, you enter its name as an operand.

Example:

```
MOV DX,BAZ
MOV CL,FIELD2
SHR DX,CL
```

FIELD2 is now right shifted, ready for access.

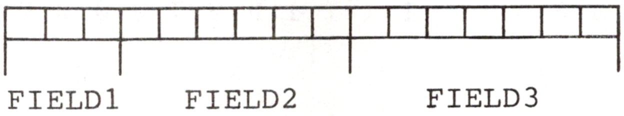
WIDTH

WIDTH <record-fieldname>
WIDTH <record>

When a <record-fieldname> is given as the argument, WIDTH returns the width of a record field as the number of bits in the record field.

When a <record> is given as the argument, WIDTH returns the width of a record as the number of bits in the record.

Using the diagram under shift count, WIDTH can be diagrammed as:



The WIDTH of FIELD1 equals 3.
The WIDTH of FIELD2 equals 6.
The WIDTH of FIELD3 equals 7.

Example:

```
MOV CL,WIDTH FIELD2
```

The number of bits in FIELD2 is now in the count register.

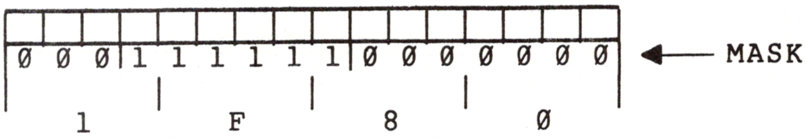
MASK

MASK <record-fieldname>

MASK accepts a field name as its only argument.

MASK returns a bit-mask defined by 1 for bit positions included by the field and 0 for bit positions not included. The value return represents the maximum value for the record when the field is masked.

Using the diagram used for shift count, MASK can be diagrammed as:



The MASK of FIELD2 equals 1F80H.

Example:

```
MOV DX,BAZ
AND DX,MASK FIELD2
```

FIELD2 is now isolated.

3.3.2 Arithmetic Operators

Eight arithmetic operators provide the common mathematical functions (add, subtract, divide, multiply, modulo, negation), plus two shift operators.

The arithmetic operators are used to combine operands to form an expression that results in a data item or an address.

Except for + and - (binary), operands must be constants.

For plus (+), one operand must be a constant.

For minus (-), the first (left) operand may be a nonconstant, or both operands may be nonconstants. The right must be a constant if the left is a constant.

*	Multiply
/	Divide
MOD	Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo). Both operands must be absolute. Example: `MOV AX,100 MOD 17` The value moved into AX will be 0FH (decimal 15).
SHR	Shift Right. SHR is followed by an integer which specifies the number of bit positions the value is to be shifted right. Example: `MOV AX,1100000B SHR 5` The value moved into AX will be 11B (03).
SHL	Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be shifted left. Example: `MOV AX,0110B SHL 5` The value moved into AX will be 011000000B (0C0H)

- (Unary Minus)	Indicates that following value is negative, as in a negative integer.
+	Add. One operand must be a constant; one may be a nonconstant.
-	Subtract the right operand from the left operand. The first (left) operand may be a nonconstant, or both operands may be nonconstants. But the right may be a nonconstant only if the left is also a nonconstant and in the same segment.

3.3.3 Relational Operators

Relational operators compare two constant operands.

If the relationship between the two operands matches the operator, FFFFH is returned.

If the relationship between the two operands does not match the operator, a zero is returned.

Relational operators are most often used with conditional directives and conditional instructions to direct program control.

EQ	Equal. Returns true if the operands equal each other.
NE	Not Equal. Returns true if the operands are not equal to each other.
LT	Less Than. Returns true if the left operand is less than the right operand.
LE	Less than or Equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater Than. Returns true if the left operand is greater than the right operand.
GE	Greater than or Equal. Returns true if the left operand is greater than or equal to the right operand.

3.3.4 Logical Operators

Logical operators compare two constant operands bitwise.

Logical operators compare the binary values of corresponding bit positions of each operand to evaluate the logical relationship defined by the logical operator.

Logical operators can be used two ways:

1. To combine operands in a logical relationship. In this case, all bits in the operands will have the same value (either 0000 or FFFFH). In fact, it is best to use these values for true (FFFFH) and false (0000) for the symbols you will use as operands, because in conditionals anything nonzero is true.

2. In bitwise operations. In this case, the bits are different, and the logical operators act the same as the instructions of the same name.

NOT	Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.
AND	Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values.
OR	Logical OR. Returns true if either operator is true or if both are true. Returns false if both operators are false. Both operands must be absolute values.
XOR	Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operators are true or if both operators are false. Both operands must be absolute values.

3.3.5 Expression Evaluation: Precedence Of Operators

Expressions are evaluated higher precedence operators first, then left to right for equal precedence operators.

Parentheses can be used to alter precedence.

For example:

```
MOV AX,101B SHL 2*2   =  MOV AX,00101000B

MOV AX,101B SHL (2*2) = MOV AX,01010000B
```

SHL and * are equal precedence. Therefore, their functions are performed in the order the operators are encountered (left to right).

Precedence of Operators

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside: parentheses ( )
   angle brackets < >
   square brackets [ ]
   Structure variable operand: <variable>.<field>

2. Segment override operator: colon (:)

3. PTR, OFFSET, SEG, TYPE, THIS

4. HIGH, LOW

5. *, /, MOD, SHL, SHR

6. +, - (both unary and binary)

7. EQ, NE, LT, LE, GT, GE

8. Logical NOT

9. Logical AND

10. Logical OR, XOR

11. SHORT,.TYPE

Contents

Chapter 4 Action: Instructions and Directives

4.1 Instructions 4-2
4.2 Directives 4-3
4.2.1 Memory Directives 4-5
4.2.2 Conditional Directives 4-37
4.2.3 Macro Directives 4-41
Repeat Directives 4-49
Special Macro Operators 4-53
4.2.4 Listing Directives 4-57

CHAPTER 4

ACTION: INSTRUCTIONS AND DIRECTIVES

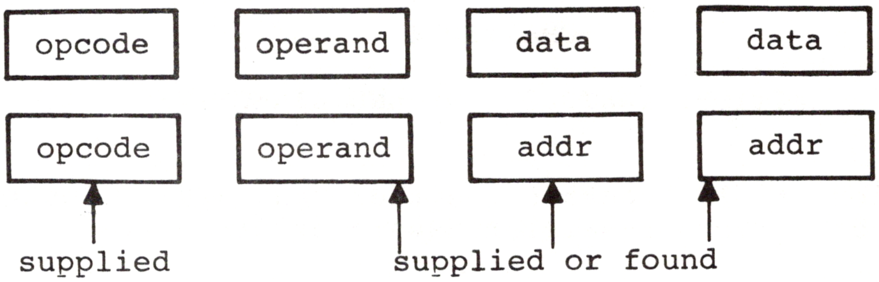
The action field contains either an 8086 instruction mnemonic or a Macro Assembler assembler directive.

Following a name field entry (if any), action field entries may begin in any column. Specific spacing is not required. The only benefit of consistent spacing is improved readability. If a statement does not have a name field entry, the action field is the first entry.

The entry in the action field either directs the processor to perform a specific function or directs the assembler to perform one of its functions.

4.1 INSTRUCTIONS

Instructions tell the command processor to perform some action. An instruction may have the data and/or addresses it needs built into it, or data and/or addresses may be found in the expression part of an instruction. For example:



supplied = part of the instruction

found = assembler inserts data and/or address from the information provided by expressions in instruction statements.

(opcode equates to the binary code for the action of an instruction)

Note that this manual does not contain detailed descriptions of the 8086 instruction mnemonics and their characteristics. For this, you will need to consult other texts. The following texts are recommended:

1. Morse, Stephen P. The 8086 Primer. Rochelle Park, NJ: Hayden Publishing Co., 1980.
2. Rector, Russell and George Alexy. The 8086 Book. Berkeley, CA: Osbourne/McGraw-Hill, 1980.
3. The 8086 Family User's Manual. Santa Clara, CA: Intel Corporation, 1980.

Appendix C contains both an alphabetical listing and a grouped listing of the instruction mnemonics. The alphabetical listing shows the full name of the instruction. Following the alphabetical list is a list that groups the instruction mnemonics by the number and type of arguments they take. Within each group, the instruction mnemonics are arranged alphabetically.

4.2 DIRECTIVES

Directives give the assembler directions and information about input and output, memory organization, conditional assembly, listing and cross-reference control, and definitions.

The directives have been divided into groups by the function they perform. Within each group, the directives are described alphabetically.

The groups are:

Memory Directives
Directives in this group are used to organize memory. Because there is no "miscellaneous" group, the memory directives group contains some directives that do not, strictly speaking, organize memory (for example, COMMENT).

Conditional Directives
Directives in this group are used to test conditions of assembly before proceeding with assembly of a block of statements. This group contains all of the IF (and related) directives.

Macro Directives
Directives in this group are used to create blocks of code called macros. This group also includes some special operators and directives that are used only inside macro blocks. The repeat directives are considered macro directives for descriptive purposes.

Listing Directives
Directives in this group are used to control the format and, to some extent, the content of listings that the assembler produces.

Appendix B contains a table of assembler directives, also grouped by function. Below is an alphabetical list of all the directives that Macro Assembler supports:

ASSUME
COMMENT
.CREF
DB
DD
DQ
DT
DW
ELSE
END
ENDIF
ENDM
ENDP
ENDS
EQU
EVEN
EXITM
EXTERN
GROUP
IF
IFB
IFDEF
IFDIF
IFE
IFIDN
IFNB
IFNDEF
IF1
IF2
IRP
IRPC
LABEL
.LALL
.LFCOND
.LIST
MACRO
NAME
ORG
%OUT
PAGE
PROC
PUBLIC
PURGE
.RADIX
RECORD
REPT
.SALL
SEGMENT
.SFCOND
STRUC
SUBTTL
.TFCOND
TITLE
.XALL
.XCREF
.XLIST

4.2.1 Memory Directives

ASSUME

ASSUME <seg-reg>:<seg-name>[,...]

or

ASSUME NOTHING

ASSUME tells the assembler that the symbols in the segment or group can be accessed using this segment register. When the assembler encounters a variable, it automatically assembles the variable reference under the proper segment register. You may enter from 1 to 4 arguments to ASSUME.

The valid <seg-reg> entries are:

CS, DS, ES, and SS.

The possible entries for <seg-name> are:

1. The name of a segment declared with the SEGMENT directive
2. The name of a group declared with the GROUP directive
3. An expression: either SEG <variable-name> or SEG <label-name> (see SEG operator, Section 3.3)
4. The key word NOTHING. ASSUME NOTHING cancels all register assignments made by a previous ASSUME statement

If ASSUME is not used or if NOTHING is typed for <seg-name>, each reference to variables, symbols, labels, and so forth in a particular segment must be prefixed by a segment register. For example, type DS:FOO instead of simply FOO.

Example:

```
ASSUME DS:DATA,SS:DATA,CS:CGROUP,ES:NOTHING
```

COMMENT

`COMMENT<delim><text><delim>`

The first non-blank character encountered after COMMENT is the delimiter. The following <text> comprises a comment block which continues until the next occurrence of <delimiter>.

COMMENT permits you to enter comments about your program without entering a semicolon (;) before each line.

If you use COMMENT inside a macro block, the comment block will not appear on your listing unless you also place the .LALL directive in your source file.

Example:

Using an asterisk as the delimiter, the format of the comment block would be:

```
COMMENT  *
any amount of text entered
here as the comment block
  .
  .
  .      *     ;return to normal mode
```

DEFINE BYTE
DEFINE WORD
DEFINE DOUBLEWORD
DEFINE QUADWORD
DEFINE TENBYTES

```
<varname>     DB     <exp>[,<exp>,...]
<varname>     DW     <exp>[,<exp>,...]
<varname>     DD     <exp>[,<exp>,...]
<varname>     DQ     <exp>[,<exp>,...]
<varname>     DT     <exp>[,<exp>,...]
```

The DEFINE directives are used to define variables or to initialize portions of memory.

If the optional <varname> is entered, the DEFINE directives define the name as a variable. If <varname> has a colon, it becomes a NEAR label instead of a variable. (See also, Section 2.1, "Labels," and Section 2.2, "Variables.")

The DEFINE directives allocate memory in units specified by the second letter of the directive (each DEFINE directive may allocate one or more of its units at a time):

DB allocates one byte (8 bits)
DW allocates one word (2 bytes)
DD allocates two words (4 bytes)
DQ allocates four words (8 bytes)
DT allocates ten bytes

<exp> may be one or more of the following:

1. A constant expression

2. The character ? for indeterminate initialization. Usually the ? is used to reserve space without placing any particular value into it. (It is the equivalent of the DS pseudo-op in MACRO-80).

3. An address expression (for DW and DD only)

4. An ASCII string (longer than two characters for DB only)

5. <exp>DUP(?)
   When this type of expression is the only argument to a define directive, the define directive produces an uninitialized data block. This expression with the ? instead of a value results in a smaller object file because only the segment offset is changed to reserve space.

6. <exp> DUP(<exp>[,...])
   This expression, like item 5, produces a data block, but initialized with the value of the second <exp>. The first <exp> must be a constant greater than zero and must not be a forward reference.

Example - Define Byte (DB):

```
NUM_BASE    DB      16
FILLER      DB      ?                ;initialize with
                                     ;indeterminate value
ONE_CHAR    DB      'M'
MULT_CHAR   DB      'TOM JEROME EDWARD BOB DEAN'
MSG         DB      'MSGTEST',13,10  ;message, carriage return
                                     ;and linefeed
BUFFER      DB      10 DUP(?)        ;indeterminate block
TABLE       DB      100 DUP(5 DUP(4),7)
                                     ;100 copies of bytes
                                     ;with values 4,4,4,4,4,7
NEW_PAGE    DB      0CH              ;form feed character
ARRAY       DB      1,2,3,4,5,6,7
```

Example - Define Word (DW):

```
ITEMS       DW      TABLE,TABLE+10,TABLE+20
SEGVAL      DW      0FFF0H
BSIZE       DW      4 * 128
LOCATION    DW      TOTAL + 1
AREA        DW      100 DUP(?)
CLEARED     DW      50 DUP(0)
SERIES      DW      2 DUP(2,3 DUP(BSIZE))
            ;two words with the byte values
            ;2,BSIZE,BSIZE,BSIZE,2,BSIZE,BSIZE,BSIZE
DISTANCE    DW      START_TAB -END_TAB
            ;difference of two labels is a constant
```

Example - Define Doubleword (DD):

```
DBPTR            DD      TABLE          ;16-bit OFFSET,
                                        ;then 16-bit
                                        ;SEG base value
SEC_PER_DAY      DD      60*60*24       ;arithmetic is performed
                                        ;by the assembler
LIST             DD      'XY',2 DUP(?)
HIGH             DD      4294967295     ;maximum
FLOAT            DD      6.735E2        ;floating point
```

Example - Define Quadword (DQ):

```
LONG_REAL       DQ      3.141597                 ;decimal makes
                                                 ;it real
STRING          DQ      'AB'                     ;no more than 2
                                                 ;characters
HIGH            DQ      18446744073709661615     ;maximum
LOW             DQ      -18446744073709661615    ;minimum
SPACER          DQ      2 DUP(?)                 ;uninit.data
FILLER          DQ      1 DUP(?,?)               ;initalized w_/
                                                 ;indeterminate
                                                 ;value
HEX_REAL        DQ      0FDCBA9A98765432105R
```

Example - Define Tenbytes (DT):

```
ACCUMULATOR     DT      ?
STRING          DT      'CD'            ;no more than 2
                                        ;characters
PACKED_DECIMAL DT       1234567890
FLOATING_POINT DT       3.1415926
```

END

```
END     [<exp>]
```

The END statement specifies the end of the program.

If <exp> is present, it is the start address of the program. If several modules are to be linked, only the main module may specify the start of the program with the END <exp> statement.

If <exp> is not present, then no start address is passed to MS-LINK for that program or module.

Example:

```
END     START    ;START is a label somewhere in the
                 ;program
```

EQU

```
<name>      EQU     <exp>
```

EQU assigns the value of <exp> to <name>. If <exp> is an external symbol, an error is generated. If <name> already has a value, an error is generated. If you want to be able to redefine a <name> in your program, use the equal sign (=) directive instead.

In many cases, EQU is used as a primitive text substitution, like a macro.

<exp> may be any one of the following:

1. A symbol. <name> becomes an alias for the symbol in <exp>. Shown as an Alias in the symbol table.
2. An instruction name. Shown as an Opcode in the symbol table.
3. A valid expression. Shown as a Number or L (label) in the symbol table.
4. Any other entry, including text, index references, segment prefix and operands. Shown as Text in the symbol table.

Example:

```
FOO     EQU     BAZ           ;must be defined in this
                              ;module or an error
                              ;results
B       EQU     [BP+8]        ;index reference (Text)
P8      EQU     DS:[BP+8]     ;segment prefix
                              ;and operand (Text)
CBD     EQU     AAD           ;an instruction name
                              ;(Opcode)
ALL     EQU     DEFREC<2,3,4> ;DEFREC = record name
                              ;<2,3,4> = initial values
                              ;for fields of record
EMP     EQU     6             ;constant value
FPV     EQU     6.3E7         ;floating point (text)
```

Equal Sign

```
<name>     =     <exp>
```

<exp> must be a valid expression. It is shown as a Number or L (label) in the symbol table (same as <exp> type 3 under the EQU directive above).

The equal sign (=) allows the user to set and to redefine symbols. The equal sign is like the EQU directive, except the user can redefine the symbol without generating an error. Redefinition may take place more than once, and redefinition may refer to a previous definition.

Example:

```
FOO      =       5       ;the same as FOO EQU 5
FOO      EQU     6;      ;error, FOO cannot be
                         ;redefined by EQU
FOO      =       7       ;FOO can be redefined
                         ;only by another =
FOO      =       FOO+3   ;redefinition may refer
                         ;to a previous definition
```

EVEN

EVEN

The EVEN directive causes the program counter to go to an even boundary; that is, to an address that begins a word. If the program counter is not already at an even boundary, EVEN causes the assembler to add a NOP instruction so that the counter will reach an even boundary.

An error results if EVEN is used with a byte-aligned segment.

Example:

Before: The PC points to 0019 hex (25 decimal)

EVEN

After: The PC points to 1A hex (26 decimal)
0019 hex now contains a NOP instruction

EXTRN

EXTRN <name>:<type>[,...]

<name> is a symbol that is defined in another module. <name> must have been declared PUBLIC in the module where <name> is defined.

<type> may be any one of the following, but must be a valid type for <name>:

1. BYTE, WORD, or DWORD
2. NEAR or FAR for labels or procedures (defined under a PROC directive)
3. ABS for pure numbers (implicit size is WORD, but includes BYTE)

Unlike the 8080 assembler, placement of the EXTRN directive is significant. If the directive is given with a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, place the directive outside all segments, then use either

```
ASSUME <seg-reg>:SEG <name>
```

or an explicit segment prefix.

NOTE

If a mistake is made and the symbol is not in the segment, MS-LINK will take the offset relative to the given segment, if possible. If the real segment is less than 64K bytes away from the reference, MS-LINK may find the definition. If the real segment is more than 64K bytes away, MS-LINK will fail to make the link between the reference and the definition and will return an error message.

Example:

```
In Same Segment:        In Another Segment:
------------------------------------------------------------
In Module 1:            In Module 1:

CSEG    SEGMENT         CSEGA   SEGMENT
        PUBLIC TAGN                     PUBLIC TAGF
         .                       .
         .                       .
         .                       .
TAGN:                   TAGF:
         .                       .
         .                       .
         .                       .
CSEG    ENDS            CSEGA   ENDS

In Module 2:            In Module 2:

CSEG    SEGMENT                 EXTRN TAGF:FAR
        EXTRN TAGN:NEAR         CSEGB   SEGMENT
         .                       .
         .                       .
         .                       .
        JMP TAGN                        JMP TAGF
CSEG    ENDS            CSEGB   ENDS
```

GROUP

```
<name>     GROUP     <seg-name>[,...]
```

The GROUP directive collects the segments named after GROUP (<seg-name>s) under one name. The GROUP is used by MS-LINK so that it knows which segments should be loaded together (the order the segments are named here does not influence the order in which the segments are loaded. The order in which the segments are loaded is determined by the CLASS designation of the SEGMENT directive, or by the order you name object modules in response to the MS-LINK Object Module: prompt).

All segments in a GROUP must fit into 64K bytes of memory. The assembler does not check this at all, but leaves the checking to MS-LINK.

<seg-name> may be one of the following:

1. A segment name, assigned by a SEGMENT directive. The name may be a forward reference.

2. An expression: either SEG <var>
   or SEG <label>
   Both of these entries resolve themselves to a segment name (see SEG operator, Section 3.3).

Once you have defined a group name, you can use the name:

1. As an immediate value:

   ```
   MOV AX,DGROUP
   MOV DS,AX
   ```

   DGROUP is the paragraph address of the base of DGROUP.

2. In ASSUME statements:

   ```
   ASSUME DS:DGROUP
   ```

   The DS register can now be used to reach any symbol in any segment of the group.

3. As an operand prefix (for segment override):

```
MOV BX,OFFSET DGROUP:FOO
DW  DGROUP:FOO
DD  DGROUP:FOO
```

DGROUP: forces the offset to be relative to DGROUP, instead of to the segment in which FOO is defined.

Example (Using GROUP to combine segments):

In Module A:

```
CGROUP  GROUP   XXX,YYY
XXX     SEGMENT
        ASSUME  CS:CGROUP
          .
          .
          .
XXX     ENDS
YYY     SEGMENT
          .
          .
          .
YYY     ENDS
        END
```

In Module B:

```
CGROUP  GROUP   ZZZ
ZZZ     SEGMENT
        ASSUME  CS:CGROUP
        _ .
        _ .
        _ .
ZZZ     ENDS
        END
```

INCLUDE

INCLUDE <filename>

The INCLUDE directive inserts source code from an alternate assembly language source file into the current source file during assembly. Use of the INCLUDE directive eliminates the need to repeat an often-used sequence of statements in the current source file.

The <filename> is any valid file specification for the operating system. If the device designation is other than the default, the source filename specification must include it. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the INCLUDE directive statement. When end-of-file is reached, assembly resumes with the next statement following the INCLUDE directive.

Nested INCLUDES are allowed (the file inserted with an INCLUDE statement may contain an INCLUDE directive). However, this is not a recommended practice with small systems because of the amount of memory that may be required.

The file specified must exist. If the file is not found, an error is displayed, and the assembly aborts.

On a Macro Assembler listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See Section 5.5, "Formats of Listings and Symbol Tables," for a description of listing file formats.

Example:

```
INCLUDE ENTRY
INCLUDE B:RECORD.TST
```

LABEL

```
<name>      LABEL     <type>
```

By using LABEL to define a <name>, you cause the assembler to associate the current segment offset with <name>.

The item is assigned a length of 1.

<type> varies depending on the use of <name>. <name> may be used for code or for data.

1. For code (for example, as a JMP or CALL operand):

   <type> may be either NEAR or FAR. <name> cannot be used in data manipulation instructions without using a type override.

   If you wish, you can define a NEAR label using the <name>: form (the LABEL directive is not used in this case). If you are defining a BYTE or WORD NEAR label, you can place the <name>: in front of a Define directive.

   When using a LABEL for code (NEAR or FAR), the segment must be addressable through the CS register.

   Example - For Code:

```
SUBRTF  LABEL   FAR
SUBRT:  (first instruction)     ;colon = NEAR label
```

2. For data:

<type> may be BYTE, WORD, DWORD, <structure-name>, or <record-name>. When STRUC or RECORD name is used, <name> is assigned the size of the structure or record.

Example - For Data:

```
BARRAY  LABEL  BYTE
ARRAY   DW     100 DUP(0)
          .
          .
          .
        ADD    AL,BARRAY[99]  ;ADD 100th byte to AL
        ADD    AX,ARRAY[98]   ;ADD 50th word to AX
```

By defining the array two ways, you can access entries either by byte or by word. Also, you can use this method for STRUC. It allows you to place your data in memory as a table, and to access it without the offset of the STRUC.

Defining the array two ways also permits you to avoid using the PTR operator. The double defining method is especially effective if you access the data different ways. It is easier to give the array a second name than to remember to use PTR.

NAME

NAME <module-name>

<module-name> must not be a reserved word. The module name may be any length, but Macro Assembler uses only the first six characters and truncates the rest.

The module name is passed to MS-LINK, but otherwise has no significance for the assembler. Macro Assembler does check to see if more than one module name has been declared.

Every module has a name. Macro Assembler derives the module name from:

1. A valid NAME directive statement
2. If the module does not contain a NAME statement, Macro Assembler uses the first six characters of a TITLE directive statement. The first six characters must be legal as a name.

Example:

```
NAME CURSOR
```

ORG

```
ORG     <exp>
```

The location counter is set to the value of <exp>, and the assembler assigns generated code starting with that value.

All names used in <exp> must be known on pass 1. The value of <exp> must either evaluate to an absolute or must be in the same segment as the location counter.

Example:

```
ORG     120H      ;2-byte absolute value
                  ;maximum=0FFFFH
ORG     $+2       ;skip two bytes
```

Example - ORG to a boundary (conditional):

```
CSEG    SEGMENT PAGE
BEGIN   =       $
        .
        .
        .
IF ($-BEGIN) MOD 256    ;if not already on
                        ;256-byte boundary
        ORG ($-BEGIN)+256-(($-BEGIN) MOD 256)
ENDIF
```

See Section 4.2.2, "Conditional Directives," for an explanation of conditional assembly.

PROC

```
<procname>      PROC      [NEAR]
                      or   [FAR]
                 .
                 .
                 .
                 RET
<procname>      ENDP
```

The default, if no operand is specified, is NEAR. Use FAR if:

1. The procedure name is an operating system entry point
2. The procedure will be called from code which has another ASSUME CS value

Each PROC block should contain a RET statement.

The PROC directive serves as a structuring device to make your programs more understandable.

The PROC directive, through the NEAR/FAR option, informs CALLs to the procedure to generate a NEAR or a FAR CALL, and RETs to generate a NEAR or a FAR RET. PROC is used, therefore, for coding simplification so that the user does not have to worry about NEAR or FAR for CALLs and RETs.

A NEAR CALL or RETURN changes the IP but not the CS register. A FAR CALL or RETURN changes both the IP and the CS registers.

Procedures are executed either in line, from a JMP, or from a CALL.

PROCs may be nested, which means that they are put in line.

Combining the PUBLIC directive with a PROC statement (both NEAR and FAR), permits you to make external CALLs to the procedure or to make other external references to the procedure.

Example:

```
        PUBLIC  FAR_NAME
FAR_NAME       PROC    FAR
        CALL    NEAR_NAME
        RET
FAR_NAME       ENDP

        PUBLIC  NEAR_NAME
NEAR_NAME      PROC    NEAR
        .
        .
        .
        RET
NEAR_NAME      ENDP
```

The second subroutine above can be called directly from a NEAR segment (that is, a segment addressable through the same CS and within 64K):

```
    CALL NEAR_NAME
```

A FAR segment (that is, any other segment that is not a NEAR segment) must call to the first subroutine, which then calls the second (an indirect call):

```
    CALL FAR_NAME
```

PUBLIC

```
PUBLIC     <symbol>[,...]
```

Place a PUBLIC directive statement in any module that contains symbols you want to use in other modules without defining the symbol again. PUBLIC makes the listed symbol(s), which are defined in the module where the PUBLIC statement appears, available for use by other modules to be linked with the module that defines the symbol(s). This information is passed to MS-LINK.

<symbol> may be a number, a variable, a label (including PROC labels).

<symbol> may not be a register name or a symbol defined (with EQU) by floating point numbers or by integers larger than two bytes.

Example:

```
        PUBLIC  GETINFO
GETINFO PROC    FAR
        PUSH    BP        ;save caller's register
        MOV     BP,SP     ;get address parameters
                          ;body of subroutine
        POP     BP        ;restore caller's reg
        RET               ;return to caller
GETINFO ENDP
```

Example - illegal PUBLIC:

```
        PUBLIC PIE_BALD,HIGH_VALUE
PIE_BALD       EQU     3.1416
HIGH_VALUE EQU 999999999
```

.RADIX

.RADIX <exp>

The default input base (or radix) for all constants is decimal. The .RADIX directive permits you to change the input radix to any base in the range 2 to 16.

<exp> is always in decimal radix, regardless of the current input radix.

Example:

```
        MOV     BX,0FFH
        .RADIX  16
        MOV     BX,0FF
```

The two MOVs in this example are identical.

The .RADIX directive does not affect the generated code values placed in the .OBJ, .LST, or .CRF output files.

The .RADIX directive does not affect the DD, DQ, or DT directives. Numeric values entered in the expression of these directives are always evaluated as decimal unless a data type suffix is appended to the value.

Example:

```
        .RADIX 16
NUM_HAND        DT      773  ;773 = decimal
HOT_HAND        DQ      773Q ;773 = octal here only
COOL_HAND       DD      773H ;now 773 = hexadecimal
```

RECORD

```
<recordname>      RECORD    <fieldname>:<width>[=<exp>],[...]
```

<fieldname> is the name of the field. <width> specifies the number of bits in the field defined by <fieldname>. <exp> contains the initial (or default) value for the field. Forward references are not allowed in a RECORD statement.

<fieldname> becomes a value that can be used in expressions. When you use <fieldname> in an expression, its value is the shift count to move the field to the far right. Using the MASK operator with the <fieldname> returns a bit mask for that field.

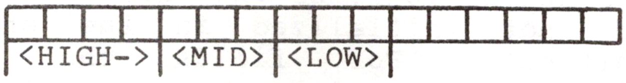
<width> is a constant in the range 1 to 16 that specifies the number of bits contained in the field defined by <fieldname>. The WIDTH operator returns this value. If the total width of all declared fields is larger than 8 bits, then the assembler uses two bytes. Otherwise, only one byte is used.

The first field you declare goes into the most significant bits of the record. Successively declared fields are placed in the succeeding bits to the right. If the fields you declare do not total exactly 8 bits or exactly 16 bits, the entire record is shifted right so that the last bit of the last field is the lowest bit of the record. Unused bits will be in the high end of the record.

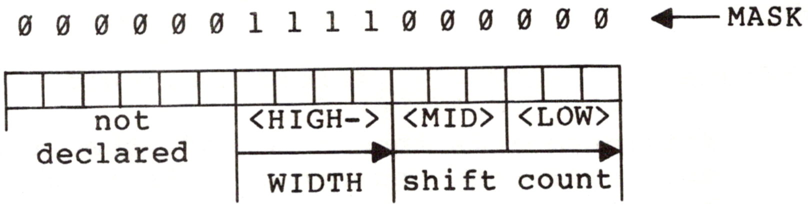
Example:

```
FOO RECORD HIGH:4,MID:3,LOW:3
```

Initially, the bit map would be:



Total bits >8 means use a word; but total bits <16 means right shift, place undeclared bits at high end of word. Thus:



<exp> contains the initial value for the field. If the field is at least 7 bits wide, you can use an ASCII character as the <exp>.

Example:

```
HIGH:7='Q'
```

To initialize records, use the same method used for DB. The format is:

```
[<name>] <recordname> <[exp][,...]>
```

or

```
[<name>] <recordname> [<exp> DUP(<[exp][,...]>)
```

The name is optional. When given, name is a label for the first byte or word of the record storage area.

The recordname is the name used as a label for the RECORD directive.

The [exp] (both forms) contains the values you want placed into the fields of the record. In the latter case, the parentheses and angle brackets are required only around the second [exp] (following DUP). If [exp] is left blank, either the default value applies (the value given in the original record definition), or the value is indeterminate (when not initialized in the original record definition). For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields.

For example:

```
FOO <,,7>
```

From the previous example, the 7 would be placed into the LOW field of the record FOO. The fields

HIGH and MID would be left as declared (in this case, uninitialized).

Records may be used in expressions (as an operand) in the form:

```
recordname<[value[,...]]>
```

The value entry is optional. The angle brackets must be coded as shown, even if the optional values are not given. A value entry is the value to be placed into a field of the record. For fields that are already initialized to values you want, place consecutive commas to skip over (use the default values of) those fields, as shown above.

Example:

```
FOO     RECORD  HIGH:5,MID:3,LOW:3
          .
          .
          .
BAX     FOO     <>  ;leave undeterminate here
JANE    FOO     10 DUP(<16,8>)  ;HIGH=16,MID=8,
                ;LOW=?
          .
          .
          .
        MOV     DX,OFFSET JANE[2]
                ;get beginning record address
        AND     DX,MASK MID
        MOV     CL,MID
        SHR     DX,CL
        MOV     CL,WIDTH MID
```

SEGMENT

```
<segname>      SEGMENT [<align>] [<combine>] [<'class'>]
                  .
                  .
                  .
<segname>      ENDS
```

At runtime, all instructions that generate code and data are in (separate) segments. Your program may be a segment, part of a segment, several segments, parts of several segments, or a combination of these. If a program has no SEGMENT statement, an MS-LINK error (invalid object) will result at link time.

The <segment name> must be a unique, legal name. The segment name must not be a reserved word.

<align> may be PARA (paragraph - default), BYTE, WORD, or PAGE.

<combine> may be PUBLIC, COMMON, AT <exp>, STACK, MEMORY, or no entry (which defaults to not combinable, called Private in the Microsoft LINK section of the Macro Assembler Manual).

<class> name is used to group segments at link time.

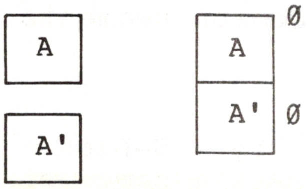
All three operands are passed to MS-LINK.

The alignment type tells the Linker on what kind of boundary you want the segment to begin. The first address of the segment will be, for each alignment type:

```
PAGE - address is xxx00H (low byte is 0)
PARA - address is xxxx0H (low nibble is 0)
    bit map - |x|x|x|x|0|0|0|0|
WORD - address is xxxxeH (e=even number;low bit
    is 0)
    bit map - |x|x|x|x|x|x|x|0|
BYTE - address is xxxxxH (place anywhere)
```

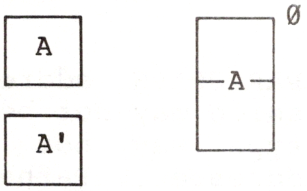
The combine type tells MS-LINK how to arrange the segments of a particular class name. The segments are mapped as follows for each combine type:

None (not combinable or Private)



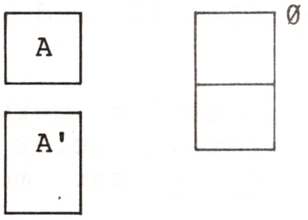
Private segments are loaded separately and remain separate. They may be physically contiguous but not logically, even if the segments have the same name. Each private segment has its own base address.

Public and Stack



Public segments of the same name and class name are loaded contiguously. Offset is from beginning of first segment loaded through last segment loaded. There is only one base address for all public segments of the same name and class name. (Combine type stack is treated the same as public. However, the Stack Pointer is set to the first address of the first stack segment. MS-LINK requires at least one stack segment.)

Common



Common segments of the same name and class name are loaded overlapping one another. There is only one base address for all common segments of the same name. The length of the common area is the length of the longest segment.

Memory

The memory combine type causes the segment(s) to be placed as the highest segments in memory. The first memory combinable segment encounter is placed as the highest segment in memory. Subsequent segments are treated the same as Common segments.

NOTE

> This feature is not supported by MS-LINK. MS-LINK treats Memory segments the same as Public segments.

AT <exp>

The segment is placed at the PARAGRAPH address specified in <exp>. The expression may not be a forward reference. Also, the AT type may not be used to force loading at fixed addresses. Rather, the AT combine type permits labels and variables to be defined at fixed offsets within fixed areas of storage, such as ROM or the vector space in low memory.

NOTE

> This restriction is imposed by MS-LINK and MS-DOS.

Class names must be enclosed in quotation marks. Class names may be any legal name. Refer to Chapter 9 in the *MS-DOS User's Guide* for more discussion.

Segment definitions may be nested. When segments are nested, the assembler acts as if they are not and handles them sequentially by appending the second part of the split segment to the first. At ENDS for the split segment, the assembler takes up the nested segment as the next segment, completes it, and goes on to subsequent segments. Overlapping segments are not permitted.

For example:

```
A      SEGMENT   |      |A SEGMENT
         .       |      |    .
         .       |      |    .
         .       |      |    .
B      SEGMENT   |      |A ENDS
         .       |      |B         SEGMENT
         .       | ---> |    .
         .       |      |    .
B      ENDS      |      |    .
         .       |      |B         ENDS
         .       |      |A         SEGMENT
         .       |      |    .
A      ENDS      |      |    .
                        |    .
                        |A ENDS
```

The following arrangement is not allowed:

```
A      SEGMENT
         .
         .
B      SEGMENT
         .
         .
A      ENDS     ;This is illegal!
         .
         .
B      ENDS
```

Example:

In module A:

```
SEGA    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:SEGA
         .
         .
         .
SEGA    ENDS
        END
```

In module B:

```
SEGA    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:SEGA
         .      ;MS-LINK adds this segment to same
         .      ;named segment in module A (and
         .      ;others) if class name is the same.
SEGA    ENDS
        END
```

STRUC

```
<structurename>     STRUC
                      .
                      .
                      .
<structurename>     ENDS
```

The STRUC directive is very much like RECORD, except STRUC has a multiple byte capability. The allocation and initialization of a STRUC block are the same as for RECORDs.

Inside the STRUC/ENDS block, the Define directives (DB, DW, DD, DQ, DT) may be used to allocate space. The Define directives and Comments set off by semicolons (;) are the only statement entries allowed inside a STRUC block.

Any label on a Define directive inside a STRUC/ENDS block becomes a <fieldname> of the structure. (This is how structure fieldnames are defined.) Initial values given to fieldnames in the STRUC/ENDS block are default values for the various fields. These field values are of two types: overridable or not overridable. A simple field, a field with only one entry (but not a DUP expression), is overridable. A multiple field, a field with more than one entry, is not overridable. For example:

```
FOO     DB      1,2                ;is not
overridable
BAZ     DB      10 DUP(?)          ;is not
overridable
ZOO     DB      5                  ;is overridable
```

If the <exp> following the Define directive contains a string, it may be overridden by another string. However, if the overriding string is shorter than the initial string, the assembler will pad with spaces. If the overriding string is longer, the assembler will truncate the extra characters.

Usually, structure fields are used as operands in some expression. The format for a reference to a structure field is:

```
    <variable>.<field>
```

<variable> represents an anonymous variable, usually set up when the structure is allocated. To allocate a structure, use the structure name as a directive with a label (the anonymous variable of a structure reference) and any override values in angle brackets:

```
    FOO      STRUCTURE
                .
                .
                .
    FOO      ENDS

    GOO      FOO      <,7,,'JOE'>
```

.<field> represents a label given to a DEFINE directive inside a STRUC/ENDS block (the period must be coded as shown). The value of <field> will be the offset within the addressed structure.

Example:

To define a structure:

```
S  STRUC
FIELD1  DB      1,2             ;not overridable
FIELD2  DB      10 DUP(?)       ;not overridable
FIELD3  DB      5               ;overridable
FIELD4  DB      'DOBOSKY'       ;overridable
S       ENDS
```

The Define directives in this example define the fields of the structure, and the order corresponds to the order values are given in the initialization list when the structure is allocated. Every Define directive statement line inside a STRUC block defines a field, whether or not the field is named.

To allocate the structure:

```
DBAREA  S     <,,7,'ANDY'>     ;overrides 3rd and
4th
                               ;fields only
```

To refer to a structure:

```
        MOV     AL,[BX].FIELD3
        MOV     AL,DBAREA.FIELD3
```

4.2.2 Conditional Directives

Conditional directives allow users to design blocks of code which test for specific conditions.

All conditionals follow the format:

```
IFxxxx [argument]
 .
 .
 .
[ELSE
 .
 .
 . ]
ENDIF
```

Each IFxxxx must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a Code 8, "Not in conditional block" error.

Each conditional block may include the optional ELSE directive, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a Code 7, "Already had ELSE clause" error.

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass 1 to avoid Phase errors and incorrect evaluation. For IF and IFE the expression must involve values which were previously defined, and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

The assembler evaluates the conditional statement to TRUE (which equals any non-zero value), or to FALSE (which equals 0000H). If the evaluation matches the condition defined in the conditional statement, the assembler either assembles the whole conditional block or, if the conditional block contains the optional ELSE directive, assembles from IF to ELSE; the ELSE to ENDIF portion of the block is ignored. If the evaluation does not match, the assembler either ignores the conditional block completely or, if the conditional block contains the optional ELSE directive, assembles only the ELSE to ENDIF portion; the IF to ELSE portion is ignored.

The following is a list of Macro Assembler conditional directives: IF <exp>

If <exp> evaluates to nonzero, the statements within the conditional block are assembled.

IFE <exp>

If <exp> evaluates to 0, the statements in the conditional block are assembled.

IF1 Pass 1 Conditional

If the assembler is in pass 1, the statements in the conditional block are assembled. IF1 takes no expression.

IF2 Pass 2 Conditional

If the assembler is in pass 2, the statements in the conditional block are assembled. IF2 takes no expression.

IFDEF <symbol>

If the <symbol> is defined or has been declared External, the statements in the conditional block are assembled.

IFNDEF <symbol>

If the <symbol> is not defined or not declared External, the statements in the conditional block are assembled.

IFB <arg>

The angle brackets around <arg> are required.

If the <arg> is blank (none given) or null (two angle brackets with nothing in between, <>), the statements in the conditional block are assembled.

IFB (and IFNB) are normally used inside macro blocks. The expression following the IFB directive is typically a dummy symbol. When the macro is called, the dummy will be replaced by a parameter passed by the macro call. If the macro call does not specify a parameter to replace the dummy following IFB, the expression is blank, and the block will be assembled. (IFNB is the opposite case.) Refer to Section 4.2.3, "Macro Directives," for a full explanation.

IFNB <arg>

The angle brackets around <arg> are required.

If <arg> is not blank, the statements in the conditional block are assembled.

IFNB (and IFB) are normally used inside macro blocks. The expression following the IFNB directive is typically a dummy symbol. When the macro is called, the dummy will be replaced by a parameter passed by the macro call. If the macro call specifies a parameter to replace the dummy following IFNB, the expression is not blank, and the block will be assembled. (IFB is the opposite case.) Refer to Section 4.2.3, "Macro Directives," for a full explanation.

IFIDN <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

IFIDN (and IFDIF) are normally used inside macro blocks. The expression following the IFIDN directive is typically two dummy symbols. When the macro is called, the dummys will be replaced by parameters passed by the macro call. If the macro call specifies two identical parameters to replace the dummys, the block will be assembled. (IFDIF is the opposite case.) Refer to Section 4.2.3, "Macro Directives," for a full explanation.

IFDIF <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled.

IFDIF (and IFIDN) are normally used inside macro blocks. The expression following the IFDIF directive is typically two dummy symbols. When the macro is called, the dummys will be replaced by parameters passed by the macro call. If the macro call specifies two different parameters to replace the dummys, the block will be assembled. (IFIDN is the opposite case.)

ELSE

The ELSE directive allows you to generate alternate code when the opposite condition exists. ELSE may be used with any of the conditional directives. Only one ELSE is allowed for each IFxxxx conditional directive. ELSE takes no expression.

ENDIF

This directive terminates a conditional block. An ENDIF directive must be given for every IFxxxx directive used. ENDIF takes no expression. ENDIF closes the most recent, unterminated IF.

4.2.3 Macro Directives

The macro directives allow you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition directive or one of the repetition directives, and end with the ENDM directive. All of the macro directives may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro directives of the Macro Assembler include:

macro definition:
    MACRO

termination:
    ENDM
    EXITM

unique symbols within macro blocks:
    LOCAL

undefine a macro:
    PURGE

repetitions:
    REPT (repeat)
    IRP (indefinite repeat)
    IRPC (indefinite repeat character)

The macro directives also include some special macro operators:

& (ampersand)

;; (double semicolon)

! (exclamation mark)

% (percent sign)

Macro Definition

```
<name> MACRO [<dummy>,...]
         .
         .
         .
       ENDM
```

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the macro's definition.

<name> is like a label and conforms to the rules for forming symbols. After the macro has been defined, <name> is used to invoke the macro.

A <dummy> is formed as any other name is formed. A <dummy> is a place holder that is replaced by a parameter in a one-for-one text substitution when the macro block is used. You should include all <dummy>s used inside the macro block on this line. The number of <dummy>s is limited only by the length of a line. If you specify more than one <dummy>, they must be separated by commas. Macro Assembler interprets a series of <dummy>s the same as any list of symbol names.

NOTE

> A <dummy> is always recognized exclusively as a dummy. Even if a register name (such as AX or BH) is used as a <dummy>, it will be replaced by a parameter during expansion.

One alternative is to list no <dummy>s:

```
<name> MACRO
```

This type of macro block allows you to call the block repeatedly, even if you do not want or need to pass parameters to the block. In this case, the block will not contain any <dummy>s.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

MACRO is an extremely powerful directive. With it, you can change the value and effect of any instruction mnemonic, directive, label, variable, or symbol. When Macro Assembler evaluates a statement, it first looks at the macro table it builds during pass 1. If it sees a name there that matches an entry in a statement, it acts accordingly. (Remember: Macro Assembler evaluates macros, then instruction mnemonics/directives.)

If you want to use the TITLE, SUBTTL, or NAME directives for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, Macro Assembler will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., - MACRO, MACROS, and so on.

Calling a Macro

To use a macro, enter a macro call statement:

```
<name> [<parameter>,...]
```

<name> is the <name> of the macro block. A <parameter> replaces a <dummy> on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas, spaces, or tabs. If you place angle brackets around parameters separated by commas, the assembler will pass all the items inside the angle brackets as a single parameter. For example:

```
FOO 1,2,3,4,5
```

passes five parameters to the macro, but

```
FOO <1,2,3,4,5>
```

passes only one.

The number of parameters in the macro call statement need not be the same as the number of <dummy>s in the MACRO definition. If there are more parameters than <dummy>s, the extras are ignored. If there are fewer, the extra <dummy>s will be made null. The assembled code will include the macro block after each macro call statement.

Example:

```
GEN     MACRO   XX,YY,ZZ
        MOV     AX,XX
        ADD     AX,YY
        MOV     ZZ,AX
        ENDM
```

If you then enter a macro call statement:

```
GEN     DUCK,DON,FOO
```

the assembler generates the statements:

```
        MOV     AX,DUCK
        ADD     AX,DON
        MOV     FOO,AX
```

On your program listing, these statements will be preceded by a plus sign (+) to indicate that they came from a macro block.

End Macro

ENDM

ENDM tells the assembler that the MACRO or Repeat block is ended.

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM directive. Otherwise, the "Unterminated REPT/IRP/IRPC/MACRO" message is generated at the end of each pass. An unmatched ENDM also causes an error.

If you wish to be able to exit from a MACRO or repeat block before expansion is completed, use EXITM.

Exit Macro

EXITM

The EXITM directive is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional directive.

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

Example:

```
FOO     MACRO     X
X       =         0
        REPT      X
X       =         X+1
        IFE       X-0FFH  ;test X
        EXITM     ;if true, exit REPT
        ENDIF
        DB        X
        ENDM
        ENDM
```

LOCAL

LOCAL <dummy>[,<dummy>...]

The LOCAL directive is allowed only inside a macro definition block. A LOCAL statement must precede all other types of statements in the macro definition.

When LOCAL is executed, the assembler creates a unique symbol for each <dummy> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the assembler range from ??0000 to ??FFFF. Users should avoid the form ??nnnn for their own symbols.

Example:

```
0000                FUN    SEGMENT
                           ASSUME CS:FUN,DS:FUN
                    FOO    MACRO    NUM,Y
                           LOCAL    A,B,C,D,E
                    A:     DB       7
                    B:     DB       8
                    C:     DB       Y
                    D:     DW       Y+1
                    E:     DW       NUM+1
                           JMP      A
                           ENDM
                           FOO      0C00H,0BEH
0000     07       + ??0000:         DB       7
0001     08       + ??0001:         DB       8
0002     BE       + ??0002:         DB       0BEH
0003     00BF     + ??0003:         DW       0BEH+1
0005     0C01     + ??0004:         DW       0C00H+1
0007     EB F7    +        JMP      ??0000
                           FOO      03C0H,0FFH
0009     07       + ??0005:         DB       7
000A     08       + ??0006:         DB       8
000B     FF       + ??0007:         DB       0FFH
000C     0100     + ??0008:         DW       0FFH+1
000E     03C1     + ??0009:         DW       03C0H+1
0010     EB F7    +        JMP      ??0005
0012                FUN    ENDS
                           END
```

Notice that Macro Assembler has substituted LABEL names in the form ??nnnn for the instances of the dummy symbols.

PURGE

PURGE <macro-name>[,...]

PURGE deletes the definition of the macro(s) listed after it.

PURGE provides three benefits:

1. It frees text space of the macro body.
2. It returns any instruction mnemonics or directives that were redefined by macros to their original function.
3. It allows you to "edit out" macros from a macro library file. You may find it useful to create a file that contains only macro definitions. This method allows you to use macros repeatedly with easy access to their definitions. Typically, you would then place an INCLUDE statement in your program file. Following the INCLUDE statement, you could place a PURGE statement to delete any macros you will not use in this program.

   It is not necessary to PURGE a macro before redefining it. Simply place another MACRO statement in your program, reusing the macro name.

Example:

```
INCLUDE MACRO.LIB
PURGE   MAC1
MAC1               ;tries to invoke purged macro
                   ;returns a syntax error
```

Repeat Directives

The directives in this group allow the operations in a block of code to be repeated for the number of times you specify. The major differences between the Repeat directives and MACRO directive are:

1. MACRO gives the block a name by which to call in the code wherever and whenever needed; the macro block can be used in many different programs by simply entering a macro call statement.

2. MACRO allows parameters to be passed to the macro block when a MACRO is called; hence, parameters can be changed.

Repeat directive parameters must be assigned as a part of the code block. If the parameters are known in advance and will not change, and if the repetition is to be performed for every program execution, then Repeat directives are convenient. With the MACRO directive, you must call in the MACRO each time it is needed.

Note that each Repeat directive must be matched with the ENDM directive to terminate the repeat block.

Repeat

```
REPT <exp>
 .
 .
 .
ENDM
```

Repeat block of statements between REPT and ENDM <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains an External symbol or undefined operands, an error is generated.

Example:

```
                X       =       0
                        REPT    10      ;generates
                                        ;DB 1 - DB
10
                X       =       X+1
                        DB      X
                        ENDM
```

assembles as:

```
0000            X       =       0
                        REPT    10      ;generates
                                        ;DB 1 - DB
10
                X       =       X+1
                        DB      X
                        ENDM
0000'   01      +       DB      X
0001'   02      +       DB      X
0002'   03      +       DB      X
0003'   04      +       DB      X
0004'   05      +       DB      X
0005'   06      +       DB      X
0006'   07      +       DB      X
0007'   08      +       DB      X
0008'   09      +       DB      X
0009'   0A      +       DB      X
                        END
```

Indefinite Repeat

```
IRP <dummy>,<parameters inside angle brackets>
 .
 .
 .
ENDM
```

Parameters must be enclosed in angle brackets. Parameters may be any legal symbol, string, numeric, or character constant. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of <dummy> in the block. If a parameter is null (i.e., <>), the block is processed once with a null parameter.

Example:

```
IRP     X,<1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

This example generates the same bytes (DB 1 to DB 10) as the REPT example.

When IRP is used inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code as above, illustrates the removal of one level of brackets from the parameters:

```
FOO     MACRO     X
        IRP       Y,<X>
        DB        Y
        ENDM
        ENDM
```

When the macro call statement

```
FOO <1,2,3,4,5,6,7,8,9,10>
```

is assembled, the macro expansion becomes:

```
IRP     Y,<1,2,3,4,5,6,7,8,9,10>
DB      Y
ENDM
```

The angle brackets around the parameters will be removed, and all items are passed as a single parameter.

Indefinite Repeat Character

```
IRPC <dummy>,<string>
 .
 .
 .
ENDM
```

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block.

Example:

```
    IRPC    X,0123456789
    DB      X+1
    ENDM
```

This example generates the same code (DB 1 to DB 10) as the two previous examples.

Special Macro Operators

Several special operators can be used in a macro block to select additional assembly functions.

& Ampersand concatenates text or symbols. (The ampersand may not be used in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately by an ampersand. To form a symbol from text and a dummy, put an ampersand between them.

For example:

```
ERRGEN  MACRO    X
ERROR&X:        PUSH    BX
        MOV      BX,'&X'
        JMP      ERROR
        ENDM
```

The call ERRGEN A will then generate:

```
ERRORA: PUSH     B
        MOV      BX,'A'
        JMP      ERROR
```

In Macro Assembler, the ampersand will not appear in the expansion. One ampersand is removed each time a dummy& or &dummy is found. For complex macros, where nesting is involved, extra ampersands may be needed. You need to supply as many ampersands as there are levels of nesting.

For example:

```
Correct form                 Incorrect form

FOO     MACRO   X            FOO  MACRO   X
        IRP     Z,<1,2,3>         IRP     Z,<1,2,3>
X&&Z    DB      Z            X&Z  DB      Z
        ENDM                      ENDM
        ENDM                      ENDM
```

When called, for example, by FOO BAZ, the expansion would be (correctly in the left column, incorrectly in the right):

1. MACRO build, find <dummy>s and change to dl

```
        IRP     Z,<1,2,3>         IRP     Z,<1,2,3>
dl&Z    DB      Z            dlZ  DB      Z
        ENDM                      ENDM
```

2. MACRO expansion, substitute parameter text for dl

```
        IRP     Z,<1,2,3>         IRP     Z,<1,2,3>
BAZ&Z   DB      Z            BAZZ DB      Z
        ENDM                      ENDM
```

3. IRP build, find dummys and change to dl

```
BAZ&dl  DB      dl           BAZZ DB      dl
```

4. IRP expansion, substitute parameter text for dl

```
BAZ1    DB      1       BAZZ    DB      1
BAZ2    DB      2       BAZZ    DB      2
BAZ3    DB      3       BAZZ    DB      3

                          ;here it's an error,
                          ;multi-defined symbol
```

<text> Angle brackets cause Macro Assembler to treat the text between the angle brackets as a single literal. Placing parameters to a macro call inside angle brackets; or placing the list of parameters following the IRP directive inside angle brackets causes two results:

1. All text within the angle brackets is seen as a single parameter, even if commas are used.

2. Characters that have special functions are taken as literal characters. For example, the semicolon inside angle brackets <;> becomes a character, not the indicator that a comment follows.

One set of angle brackets is removed each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angle brackets around parameters as there are levels of nesting.

;; In a macro or repeat block, a comment preceded by two semicolons is not saved as a part of the expansion.

The default listing condition for macros is .XALL (see Section 4.2.4, "Listing Directives," below). Under the influence of .XALL, comments in macro blocks are not listed because they do not generate code.

If you decide to place the .LALL listing directive in your program, then comments inside macro and repeat blocks are saved and listed. This can be the cause of an "out of memory error." To avoid this error, place double semicolons before comments inside macro and repeat blocks, unless you specifically want a comment to be retained.

! An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

% The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference, with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must evaluate to an absolute (non-relocatable) constant.

Example:

```
PRINTE  MACRO   MSG,N
        %OUT   * MSG,N *
        ENDM
SYM1    EQU     100
SYM2    EQU     200
        PRINTE  <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

Normally, the macro call statement would cause the string (SYM1 + SYM2) to be substituted for the dummy N. The result would be:

```
        %OUT   * SYM1 + SYM2 = (SYM1 + SYM2) *
```

When the % is placed in front of the parameter, the assembler generates:

```
        %OUT   * SYM1 + SYM2 = 300 *
```

4.2.4 Listing Directives

Listing directives perform two general functions: format control and listing control. Format control directives allow the programmer to insert page breaks and direct page headings. Listing directives turn on and off the listing of all or part of the assembled file.

PAGE

```
PAGE [<length>][,<width>]
PAGE [+]
```

PAGE with no arguments or with the optional [,+] argument causes the assembler to start a new output page. The assembler puts a form feed character in the listing file at the end of the page.

The PAGE directive with either the length or width arguments does not start a new listing page.

The value of <length>, if included, becomes the new page length (measured in lines per page) and must be in the range 10 to 255. The default page length is 50 lines per page.

The value of <width>, if included, becomes the new page width (measured in characters) and must be in the range 60 to 132. The default page width is 80 characters.

The plus sign (+) increments the major page number and resets the minor page number to one. Page numbers are in the form major-minor. The PAGE directive without the + increments only the minor portion of the page number.

Example:

```
 .
 .
 .
PAGE +    ;increment major,set minor to 1
 .
 .
 .
PAGE 58,60    ;page length=58 lines,
              ;width=60 characters
```

TITLE

TITLE <text>

TITLE specifies a title to be listed on the first line of each page. The <text> may be up to 60 characters long. If more than one TITLE is given, an error results. The first six characters of the title, if legal, are used as the module name, unless a NAME directive is used.

Example:

```
TITLE PROG1 -- 1st Program
 .
 .
 .
```

If the NAME directive is not used, the module name is now PROG1--1st Program. This title text will appear at the top of every page of the listing.

SUBTITLE

SUBTTL <text>

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title. The <text> is truncated after 60 characters.

Any number of SUBTTLs may be given in a program. Each time the assembler encounters SUBTTL, it replaces the <text> from the previous SUBTTL with the <text> from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for <text>.

Example:

```
SUBTTL SPECIAL I/O ROUTINE
  .
  .
  .
SUBTTL
  .
  .
  .
```

The first SUBTTL causes the subtitle SPECIAL I/O ROUTINE to be printed at the top of every page. The second SUBTTL turns off subtitle (the subtitle line on the listing is left blank).

%OUT

%OUT <text>

The text is listed on the terminal during assembly. %OUT is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

%OUT will output on both passes. If only one printout is desired, use the IF1 or IF2 directive, depending on which pass you want displayed. See Section 4.2.2, "Conditional Directives," for descriptions of the IF1 and IF2 directives.

Example:

```
%OUT *Assembly half done*
```

The assembler will send this message to the terminal screen when encountered.

```
IF1
%OUT *Pass 1 started*
ENDIF

IF2
%OUT *Pass 2 started*
ENDIF
```

.LIST
.XLIST

.LIST lists all lines with their code (the default condition).

.XLIST suppresses all listing.

If you specify a listing file following the Listing: prompt, a listing file with all the source statements included will be printed.

When .XLIST is encountered in the source file, source and object code will not be listed. .XLIST remains in effect until a .LIST is encountered.

.XLIST overrides all other listing directives. Nothing will be listed, even if another listing directive (other than .LIST) is encountered.

Example:

```
  .
  .
  .
.XLIST     ;listing suspended here
  .
  .
  .
.LIST      ;listing resumes here
```

.SFCOND

.SFCOND suppresses portions of the listing that contain conditional false expressions.

.LFCOND

.LFCOND assures the listing of conditional expressions that evaluate false. This is the default condition.

.TFCOND

.TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch when the assembler is running. When /X is used, .TFCOND will cause false conditionals to list. When /X is not used, .TFCOND will suppress false conditionals.

.XALL

.XALL is the default.

.XALL lists source code and object code produced by a macro, but source lines which do not generate code are not listed.

.LALL

.LALL lists the complete macro text for all expansions, including lines that do not generate code. Comments preceded by two semicolons (;;) will not be listed.

.SALL

.SALL suppresses listing of all text and object code produced by macros.

.CREF
.XCREF

```
.CREF
.XCREF [<variable list>]
```

.CREF is the default condition. .CREF remains in effect until Macro Assembler encounters .XCREF.

.XCREF without arguments turns off the .CREF (default) directive. .XCREF remains in effect until Macro Assembler encounters .CREF. Use .XCREF to suppress the creation of cross-references in selected portions of the file. Use .CREF to restart the creation of a cross-reference file after using the .XCREF directive.

If you include one or more variables following .XCREF, these variables will not be placed in the listing or cross-reference file. All other cross-referencing, however, is not affected by an .XCREF directive with arguments. Separate the variables with commas.

Neither .CREF nor .XCREF without arguments takes effect unless you specify a cross-reference file when running the assembler. .XCREF <variable list> suppresses the variables from the symbol table listing regardless of the creation of a cross-reference file.

Example:

```
.XCREF CURSOR,FOO,GOO,BAZ,ZOO
        ;these variables will not be
        ;in the listing or cross-reference file
```

Contents

Chapter 5 Assembling a Macro Assembler Source File

5.1 How to Start Macro Assembler 5-1
5.1.1 Method 1: Prompts 5-2
Summary of Command Prompts 5-3
5.1.2 Method 2: Command Line 5-3
5.2 Macro Assembler Command Characters 5-4
5.3 Macro Assembler Command Prompts 5-5
5.4 Macro Assembler Command Switches 5-7
Summary of Command Switches 5-9
5.5 Formats of Listings and Symbol Tables 5-10
5.5.1 Program Listing 5-10
5.5.2 Differences Between Pass 1 and Pass 2 5-16
5.5.3 Symbol Table Format 5-17

CHAPTER 5

ASSEMBLING A MACRO ASSEMBLER SOURCE FILE

Assembling a program with Macro Assembler requires two types of commands: a command to start Macro Assembler, and answers to command prompts. In addition, four switches control alternate Macro Assembler features. Usually, you will type all the commands to Macro Assembler on the terminal keyboard. As an option, answers to the command prompts and any switches may be contained in response (batch) file. Two command characters are provided to assist you while entering assembler commands. These command characters are described in Section 5.2, "Command Characters."

5.1 HOW TO START MACRO ASSEMBLER

Macro Assembler may be started in two ways. By the first method, you type the commands in response to individual prompts. By the second method, you type all commands on the line used to start Macro Assembler.

Summary of Methods to Start Macro Assembler
```
===========================================================

  Method 1        MASM

  Method 2        MASM <source>,<object>,<listing>,
                  <cross-ref>[/switch...]

===========================================================
```

5.1.1 Method 1: Prompts

Type:

```
MASM
```

Macro Assembler will be loaded into memory. Then, Macro Assembler returns a series of four text prompts that appear one at a time. You answer the prompts as commands to Macro Assembler to perform specific tasks.

At the end of each line, you may specify one or more switches, each of which must be preceded by a forward slash (/).

The command prompts are summarized here and described in more detail in Section 5.3, "Macro Assembler Command Prompts."

Summary of Command Prompts

PROMPT	RESPONSES
Source filename [.ASM]:	List .ASM file to be assembled. (There is no default: a filename response is required.)
Object filename [source.OBJ	List filename for relocatable object code. (The default is source-filename.OBJ)
Source listing [NUL.LST]:	List filename for listing. (The default is no listing file.)
Cross reference [NUL.CRF]:	List filename for cross-reference file (used with MS-CREF to create a cross-reference listing). (The default is no cross-reference file.)

5.1.2 Method 2: Command Line

Type:

```
MASM <source>,<object>,<listing>,<cross-ref>[/switch...]
```

Macro Assembler will be loaded into memory. Then Macro Assembler immediately begins assembly. The entries following MASM are responses to the command prompts. The entry fields for the different prompts must be separated by commas.

where: <u>source</u> is the source filename

<u>object</u> is the name of the file to receive the relocatable output

<u>listing</u> is the name of the file to receive the listing

<u>cross-ref</u> is the name of the file to receive the cross-reference output

<u>/switch</u> are optional switches, which may be placed following any of the response entries (just before any of the commas or after the the <cross-ref>, as shown).

To select the default for a field, simply enter a second comma without space in between (see the example below).

Example:

```
MASM FUN,,FUN/D/X,FUN
```

This example causes Macro Assembler to be loaded, then causes the source file FUN.ASM to be assembled. Macro Assembler then outputs the relocatable object code to a file named FUN.OBJ (default caused by two commas in a row), creates a listing file named FUN.LST for both assembly passes but with false conditionals suppressed, and creates a cross-reference file named FUN.CRF. If names were not listed for listing and cross-reference, these files would not be created. If listing file switches are given but no filename, the switches are ignored.

5.2 MACRO ASSEMBLER COMMAND CHARACTERS

Macro Assembler provides two command characters.

Semicolon

Use a single semicolon (;), followed immediately by a carriage return, at any time after responding to the first prompt (from Source filename: on) to select default responses to the remaining prompts. This feature saves time and eliminates the need to enter a series of carriage returns.

NOTE

> Once the semicolon has been entered, you can no longer respond to any of the prompts for that assembly. Therefore, do not use the semicolon to skip over some prompts. For this, use the <RETURN> key.

Example:

```
Source filename [.ASM]:  FUN
Object filename [FUN.OBJ]:  ;
```

The remaining prompts will not appear, and Macro Assembler will use the default values (including no listing file and no cross-reference file).

To achieve the same result, you could type:

```
Source filename [.ASM]:  FUN ;
```

This response produces the same files as the previous example.

CONTROL-C

Use <CONTROL-C> at any time to abort the assembly. If you enter an erroneous response, such as the wrong filename or an incorrectly spelled filename, you must press <CONTROL-C> to exit Macro Assembler. You can then restart Macro Assembler. If the error has been typed and not entered, you may delete the erroneous characters, but for that line only.

5.3 MACRO ASSEMBLER COMMAND PROMPTS

Macro Assembler is commanded by entering responses to four text prompts. When you have typed a response to the current prompt, the next appears. When the last prompt has been answered, Macro Assembler begins assembly automatically without further command. When assembly is finished, Macro Assembler exits to the operating system. When the operating system prompt is displayed, Macro Assembler has finished successfully. If the assembly is unsuccessful, Macro Assembler displays the appropriate error message.

Macro Assembler prompts you for the names of source, object, listing, and cross-reference files.

All command prompts accept a file specification as a response. You may type:

A filename only

A device designation only

A filename and an extension

A device designation and filename, or

A device designation, filename, and extension.

Do not type only a filename extension.

The following is a discussion of the command prompts that are displayed when you start Macro Assembler with Method 1:

Source filename [.ASM]:

> Type the filename of your source program. Macro Assembler assumes by default that the filename extension is .ASM, as shown in square brackets in the prompt text. If your source program has any other filename extension, you must specify it along with the filename. Otherwise, the extension may be omitted.

Object filename [source.OBJ]:

> Type the filename you want to receive the generated object code. If you simply press the carriage return key when this prompt appears, the object file will be given the same name as the source file, but with the filename extension .OBJ. If you want your object file to have a different name or a different filename extension, you must type your choice in response to this prompt. If you want to

change only the filename but keep the .OBJ extension, type the filename only. To change the extension only, you must type both the filename and the extension.

Source listing [NUL.LST]:

Type the name of the file you want to receive the source listing. If you press the carriage return key, Macro Assembler does not produce this listing file. If you type a filename only, the listing is created and placed in a file with the name you type plus the filename extension .LST. You may also type your own extension.

The source listing file will contain a list of all the statements in your source program and will show the code and offsets generated for each statement. The listing will also show any error messages generated during the session.

Cross reference [NUL.CRF]:

Type the name of the file you want to receive the cross-reference file. If you press only the <RETURN> key, Macro Assembler does not produce this cross-reference file. If you type a filename only, the cross-reference file is created and placed in a file with the name you type plus the filename extension .CRF. You may also type your own extension.

The cross-reference file is used as the source file for the Microsoft CREF Cross-Reference Utility (MS-CREF). MS-CREF converts this cross-reference file into a cross-reference listing, which you can use to aid you during program debugging.

The cross-reference file contains a series of control symbols that identify records in the file. MS-CREF uses these control symbols to create a listing that shows all occurrences of every symbol in your program. The occurrence that defines the symbol is also identified.

5.4 MACRO ASSEMBLER COMMAND SWITCHES

The three Macro Assembler switches control assembler functions. Switches must be typed at the end of a prompt response, regardless of which method is used to start Macro Assembler. Switches may be grouped at the end of any one of the responses, or may be scattered at the end of several. If more than one switch is typed at the end of one response, each switch must be preceded by a forward slash (/). Do not specify only a switch as a response to a command prompt.

Switch	Function
/D	Produces a source listing on both assembler passes. The listings will, when compared, show where in the program phase errors occur and will, possibly, give you a clue to why the errors occur. The /D switch does not take effect unless you command Macro Assembler to create a source listing (type a filename in response to the Source listing: command prompt).
/O	Outputs the listing file in octal radix. The generated code and the offsets shown on the listing will all be given in octal. The actual code in the object file will be the same as if the /O switch were not given. The /O switch affects only the listing file.
/X	Suppresses the listing of false conditionals. If your program contains conditional blocks, the listing file will show the source statements, but no code if the condition evaluates false. To avoid the clutter of conditional blocks that do not generate code, use the /X switch to suppress the blocks that evaluate false from your listing. The /X switch does not affect any block of code in your file that is controlled by either the .SFCOND or .LFCOND directives.

If your source program contains the .TFCOND directive, the /X switch has the opposite effect. That is, normally the .TFCOND directive causes listing or suppressing of blocks of code that it controls. The first .TFCOND directive suppresses false conditionals, the second restores listing of false conditionals, and so on. When you use the /X switch, false conditionals are already suppressed. When Macro Assembler encounters the first .TFCOND directive, listing of false conditionals is restored. When the second .TFCOND is encountered (and the /X switch is used), false conditionals are again suppressed from the listing.

Of course, the /X switch has no effect if no listing is created. See additional discussion under the .TFCOND directive in Section 4.2.4, "Listing Directives."

The following chart illustrates the various effects of the conditional listing directives in combination with the /X switch.

Pseudo-op	No /X	/X
(none)	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	ON
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON

Summary of Command Switches

SWITCH	ACTION
/D	Produce a listing on both assembler passes.
/O	Show generated object code and offsets in octal radix on listing.
/X	Suppress the listing of false conditionals. Also used with the .TFCOND directive.

5.5 FORMATS OF LISTINGS AND SYMBOL TABLES

The source listing produced by Macro Assembler (created when you specify a filename in response to the Source listing: prompt) is divided into two parts.

The first part of the listing shows:

The line number for each line of the source file, if a cross-reference file is also being created.

The offset of each source line that generates code.

The code generated by each source line.

A plus sign (+), if the code came from a macro, or a letter C, if the code came from an INCLUDE file.

The source statement line.

The second part of the listing shows:

Macros--name and length in bytes

Structures and records--name, width and fields

Segments and groups--name, size, align, combine, and class

Symbols--name, type, value, and attributes

The number of warning errors and severe errors

5.5.1 Program Listing

The program portion of the listing is essentially your source program file with the line numbers, offsets, generated code, and (where applicable) a plus sign to indicate that the source statements are part of a macro block, or a letter C to indicate that the source statements are from a file input by the INCLUDE directive.

If any errors occur during assembly, the error message will be printed directly below the statement where the error occurred.

Part of a listing file follows this discussion, with notes explaining what the various entries represent.

The comments have been moved down one line because of format restrictions. If you print your listing on 132 column-paper, the comments shown here will easily fit on the same line as the rest of the statement.

Explanatory notes are spliced into the listing at points of special interest.

Summary of Listing Symbols

R = Linker resolves entry to left of R

E = External

---- = Segment name, group name, or segment variable used in MOV AX,<---->, DD <---->, JMP <---->, and so on.

= = Statement has an EQU or = directive

nn: = Statement contains a segment override

nn/ = REPxx or LOCK prefix instruction. Example:

```
003C  F3/ A5     REP  MOVSW ;move DS:SI to ES:DI
                            ;until CX=0
```

[xx] = DUP expression;xx is the value in parentheses following DUP; for example: DUP(?) places ?? where xx is shown here

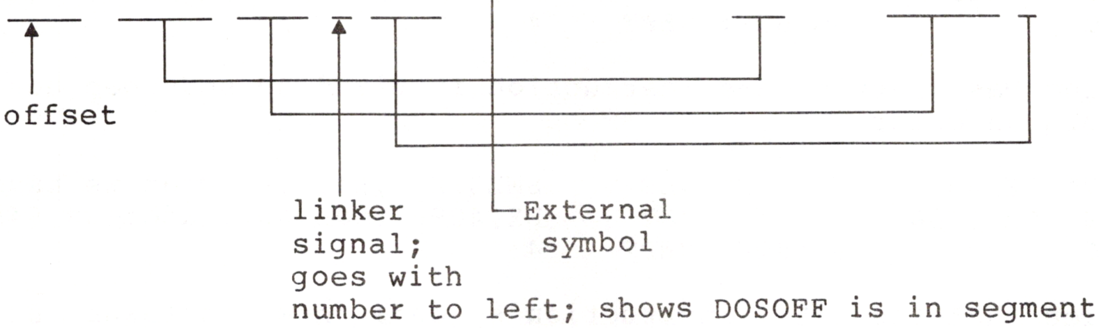
+ = Line comes from a macro expansion

C = Line comes from file named in INCLUDE directive statement

```
Microsoft Macro Assembler  1-Dec-81   PAGE 1-3

EXTX  PASCAL entry for initializing programs

                                          ;
0000            STACK       SEGMENT WORD STACK  'STACK'
= 0000          HEAPbeg     EQU     THIS BYTE
 └──── Indicates EQU or = directive
                                    ;Base of heap before init
0000      14 [              DB      20 DUP (?)
             ??◄Shows value in parentheses
             ]
               Indicates DUP expression
= 0014         SKTOP        EQU     THIS BYTE
0014           STACK        ENDS

0000            MAINSTARTUP SEGMENT  'MEMORY'
                DGROUP      GROUP   DATA,STACK<CONST,HEAP,MEMORY
                            ASSUME CS:MAINSTARTUP,DS:DGROUP,
                                   ES:DGROUP,SS:DGROUP

                            PUBLIC BEGXQQ ;Main entry

                                     ;
0000            BEGXQQ      PROC   FAR
0000   B8 ---- R            MOV    AX,DGROUP
                                   ;Get data segment value
0003   8E D8                MOV    DS,AX ;Set DS seg

0005   8C 06 0022 R         MOV    CESXQQ,ES
 │                           ▲              ╲        ╲
 │      Generated    Name       Action     Expression   Comment
Offset

                                                        .
                                                        .
                                                        .
000C   26: 8B 1E 0002       MOV     BX,ES:2  ;Highest
                                                ;paragraph
       │                                │
       └──────────────── Segment override┘
```

```
Microsoft Macro Assembler 1-Dec-81 PAGE 1-4

ENTX PASCAL entry for initializing programs

0011   2B D8               SUB    BX,AX  ;Get # paras for DS
0013   81 FB 1000          CMP    BX,4096 ;More than 64K?
0017   7E 03               JLE    SMLSTK  ;No, use what we have
0019   BB 1000             MOV    BX,4096 ;Can only address 64k

001C                 SMLSTK: +> REPT    4
                           SHL     BX,1
                               ;Convert para to offset
                           ENDM

001C   D1 E3               SHL     BX,1
                               ;Convert para to offset
001E   D1 E3               SHL     BX,1
                               ;Convert para to offset
0020   D1 E3               SHL     BX,1
                               ;Convert para to offset
0022   D1 E3               SHL     BX,1
                               ;Convert para to offset

 macro   these lines   macro                 number of
 block   from macro    directive             repetitions

0024   8B E3   MOV     SP,BX
               ;Set stack to top of memory
               .
               .
               .
0069   EA 0000        R     JMP     FAR PTR STARTmain

           signal to linker      +-segment variable

       linker resolves: indicates segment name, group name,
       or segment variable used in MOV AX,<---->;
       DD <---->; JMP <---->,etc.  (See other
       examples in this listing.)
006E      BEGXQQ            ENDP

                            .
                            .
                            .
007E      MAIN_STARTUP      ENDS

0000      ENTXCM            SEGMENT WORD 'CODE'
                ASSUME  CS:ENTXCM
                PUBLIC  ENDXQQ,DOSXQQ
```

```
Microsoft Macro Assembler  1-Dec-81    PAGE 1-5

ENTX    PASCAL entry for initializing programs


0000        STARTmain      PROC    FAR ;This code remains
0000  9A 0000 ---- E               CALL    ENTGQQ
                           ;call main program

            ;
0005        ENDXQQ         LABEL   FAR
                             ;termination entry point
0005  9A 0000 ---- E               CALL    ENDOQQ
                             ;user system termination
000A  9A 0000 ---- E               CALL    ENDYQQ
                             ;close all open files
000F  9A 0000 ---- E               CALL    ENDUQQ
                                            ;file system
                                            ;termination

0014  C7 06 0020 R 0000                         MOV     DOSOFF,0
```


offset
linker signal; goes with number to left; shows DOSOFF is in segment
External symbol

```
00 2E 0020 R               JMP     DWORD PTR DOSOFF
                                   ;return to DOS
001E        STARTmain      ENDP

                           _.
                           _.
                           _.
0037        ENTXCM         ENDS

                  END      BEGXQQ
```

5.5.2 Differences Between Pass 1 And Pass 2 Listings

If you specify the /D switch when you run Macro Assembler to assemble your file, the assembler produces a listing for both passes. The option is especially helpful for finding the source of phase errors.

The following example was taken from a source file that assembled without reporting any errors. When the source file was reassembled using the /D switch, an error was produced on pass 1, but not on pass 2 (which is when errors are usually reported).

Example:

During Pass 1 a jump with a forward reference produces:

```
0017  7E 00                JLE     SMLSTK  ;No, use what we have
 E r r o r  ---            9:Symbol not defined
0019  BB 1000              MOV     BX,4096 ;Can only address 64k
001C              SMLSTK: REPT    4
```

During Pass 2 this same instruction is fixed up and does not return an error.

```
0017  7E 03                JLE     SMLSTK  ;No, use what we have
0019  BB 1000              MOV     BX,4096 ;Can only address 64k
001C              SMLSTK: REPT    4
```

Notice that the JLE instruction's code now contains 03 instead of 00; this is a jump of 3 bytes.

The same amount of code was produced during both passes, so there was no phase error. The only difference in this case is one of content instead of size,

5.5.3 Symbol Table Format

The symbol table portion of a listing separates all "symbols" into their respective categories, showing appropriate descriptive data. This data gives you an idea how your program is using various symbolic values. and is useful when you debug.

Also, you can use a cross-reference listing, produced by MS-CREF, to help you locate uses of the various "symbols" in your program.

On the next page is a complete symbol table listing. Following the complete listing, sections from different symbol tables are shown with explanatory notes.

For all sections of symbol tables, this rule applies: if there are no symbolic values in your program for a particular category, the heading for the category will be omitted from the symbol table listing. For example, if you use no macros in your program, you will not see a macro section in the symbol table.

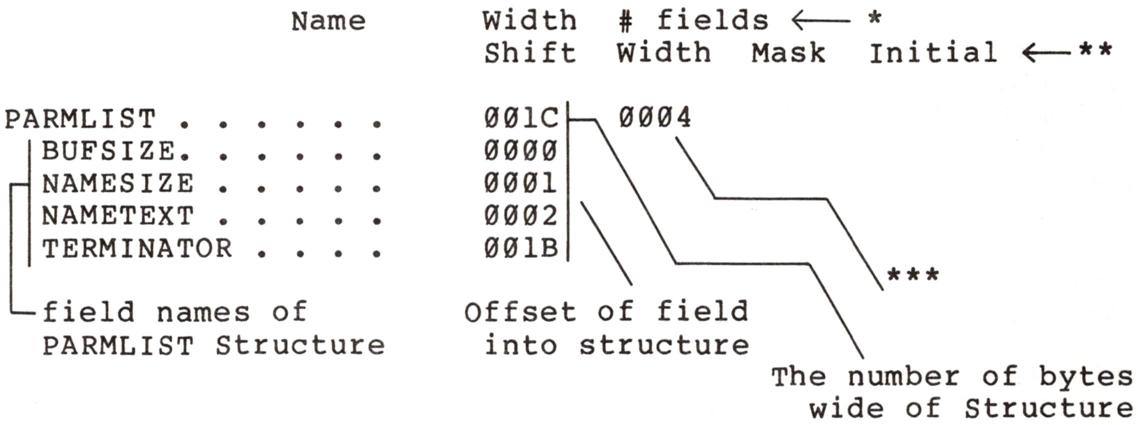
```
Microsoft Macro Assembler MACRO
Assembler      date      PAGE    Symbols-1
CALLER - SAMPLE ASSEMBLER ROUTINE (EXMP1M.ASM)

Macros:

        Name               Length

BIOSCALL . . . . . . .     0002
DISPLAY. . . . . . . .     0005
DOSCALL. . . . . . . .     0002
KEYBOARD . . . . . . .     0003
LOCATE . . . . . . . .     0003
SCROLL . . . . . . . .     0004

Structures and records:

        Name               Width   # fields
                           Shift   Width   Mask    Initial

PARMLIST . . . . . . .     001C    0004
  BUFSIZE. . . . . . .     0000
  NAMESIZE . . . . . .     0001
  NAMETEXT . . . . . .     0002
  TERMINATOR . . . . .     001B

Segments and groups:

        Name               Size    align   combine class

CSEG . . . . . . . . .     0044    PARA    PUBLIC  'CODE'
STACK. . . . . . . . .     0200    PARA    STACK   'STACK'
WORKAREA . . . . . . .     0031    PARA    PUBLIC  'DATA'

Symbols:

        Name               Type    Value   Attr

CLS. . . . . . . . . .     N PROC  0036    CSEG    Length =000E
MAXCHAR. . . . . . . .     Number  0019
MESSG. . . . . . . . .     L BYTE  001C    WORKAREA
PARMS. . . . . . . . .     L 001C  0000    WORKAREA
RECEIVR. . . . . . . .     L FAR   0000            External
START. . . . . . . . .     F PROC  0000    CSEG    Length =0036

Warning Severe
Errors  Errors
0       0
```

```
Macros:

        Name              Length ←── number of 32-byte blocks
                                     macro occupies in memory
BIOSCALL . . . . . .       0002
DISPLAY. . . . . . .       0005
DOSCALL. . . . . . .       0002
KEYBOARD . . . . . .       0003
LOCATE . . . . . . .       0003
SCROLL . . . . . . .       0004
   ↑
   │
names of macros
```

This section of the symbol table tells you the names of your macros and how big they are in 32-byte block units. In this listing, the macro DISPLAY is 5 blocks long or (5 X 32 bytes =) 160 bytes long.

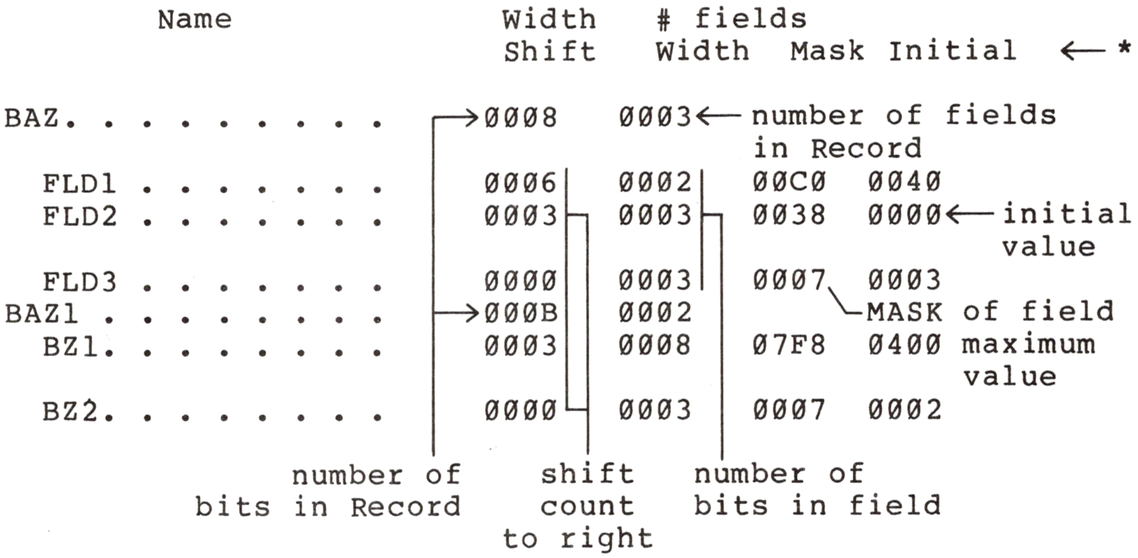
Structures and records:

Example for Structures



```
                       Name        Width  # fields <-- *
                                   Shift  Width   Mask   Initial <--**

PARMLIST . . . . . .               001C   0004
  BUFSIZE. . . . . .               0000
  NAMESIZE . . . . .               0001
  NAMETEXT . . . . .               0002
  TERMINATOR . . . .               001B
                                                              ***
  field names of                  Offset of field
  PARMLIST Structure               into structure
                                                 The number of bytes
                                                   wide of Structure
```

Example for Records



```
            Name                   Width  # fields
                                   Shift  Width  Mask  Initial  <-- *

BAZ. . . . . . . . . .          -->0008   0003<-- number of fields
                                                  in Record
  FLD1 . . . . . . . .             0006   0002    00C0   0040
  FLD2 . . . . . . . .             0003   0003    0038   0000<--initial
                                                                 value
  FLD3 . . . . . . . .             0000   0003    0007   0003
BAZ1 . . . . . . . . .          -->000B   0002        \MASK of field
  BZ1. . . . . . . . .             0003   0008    07F8   0400 maximum
                                                              value
  BZ2. . . . . . . . .             0000   0003    0007   0002

              number of      shift    number of
          bits in Record     count    bits in field
                           to right
```

* This line applies to Structure Names (begin in column 1).
** This line for fields of Records (indented).
***Number of fields in Structure.

This section lists your Structures and/or Records and their fields. The upper line of column headings applies to Structure names, Record names, and field names of Structures. The lower line of column headings applies to field names of Records.

For Structures:

Width (upper line) shows the number of bytes your Structure occupies in memory.

fields shows how many fields comprise your Structure.

For Records:

Width (upper line) shows the number of bits the Record occupies.

fields shows how many fields comprise your Record.

For Fields of Structures:

Shift shows the number of bytes the fields are offset into the Structure.

The other columns are not used for fields of Structures.

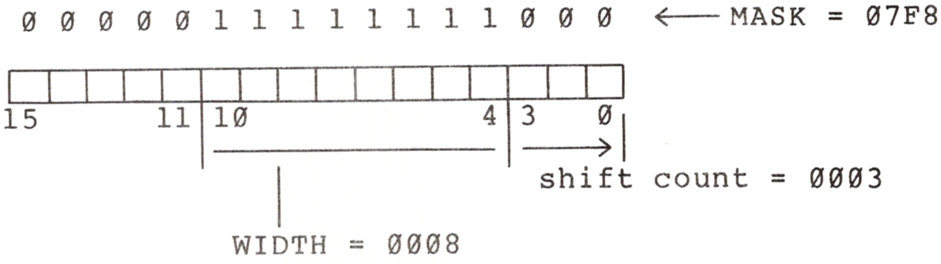
For Fields of Records:

Shift is the shift count to the right.

Width (lower line) shows the number of bits this field occupies.

Mask shows the maximum value of the record, expressed in hexadecimal, if one field is masked and ANDed (the field is set to all 1's and all other fields are set to all 0's).

Using field BZ1 of the Record BAZ1 above to illustrate:



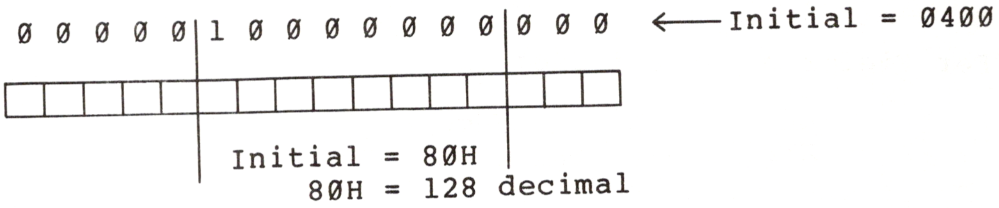
Initial shows the value specified as the initial value for the field, if any.

When naming the field, you specified:
fieldname:# = value

Fieldname is the name of the field

is the width of the field in bits

Value is the initial value you want this field to hold. The symbol table shows this value as if it is placed in the field and all other fields are masked (equal 0). Using the example and diagram from above:



Segments and groups:

```
        Name               Size     align    combine class
                                         /---called Private
                                        /  in MS-LINK manual
AAAXQQ . . . .     0000     WORD     NONE      'CODE'<--segment
DGROUP . . . .     GROUP <--------------------------group
   DATA . . . .    0024     WORD     PUBLIC    'DATA'     |
   STACK. . . .    0014     WORD     STACK     'STACK'    | segments
   CONST. . . .    0000     WORD     PUBLIC    'CONST'    |   of
   HEAP . . . .    0000     WORD     PUBLIC    'MEMORY'   |  DGROUP
   MEMORY . . .    0000     WORD     PUBLIC    'MEMORY'   |
ENTXCM . . . .     0037     WORD     NONE      'CODE'
MAIN_STARTUP .     007E     PARA     NONE      'MEMORY'
          _____    ____________________________
                   length   statement line entries
                     of
                   segment
```

For Groups:

The name of the group will appear under the Name column, beginning in column 1 with the applicable Segment names indented 2 spaces. The word Group will appear under the Size column.

For Segments:

The segment names may appear in column 1 (as here) if you do not declare them part of a group. If you declare a group, the segment names will appear indented under their group name.

For all Segments, whether a part of a group or not:

Size is the number of bytes the Segment occupies.

Align is the type of boundary where the segment begins:

PAGE = page - address is xxx00H (low byte = 0);
begins on a 256-byte boundary
PARA = paragraph - address is xxxx0H
(low nibble = 0); default

WORD = word - address is xxxxeH
(e = even number;
low bit of low byte = 0)
bit map - |x|x|x|x|x|x|x|0|

BYTE = byte - address is xxxxxH (anywhere)

Combine describes how the Microsoft LINK Linker Utility will combine the various segments. (See the Microsoft LINK Linker Utility Manual for a full description.)

Class is the class name under which MS-LINK will combine segments in memory. (See MS-LINK Linker Utility Manual and Chapter 9 of the MS-DOS User's Guide for a full description.)

```
Symbols:

            Name      Type      Value    Attr

FOO. . . . . . .      Number    0005          |
FOO1 . . . . . .      Text      1.234         |
FOO2 . . . . . .      Number    0008          |  all formed by
FOO3 . . . . . .      Alias     FOO           |  EQU or =
FOO4 . . . . . .      Text      5[BP][DI]     |  directive
FOO5 . . . . . .      Opcode                  |

Symbols:

            Name Type     Value Attr

BEGHQQ . . . . . L WORD   0012  DATA    Global
BEGOQQ . . . . . L FAR    0000      External
BEGXQQ . . . . . F PROC   0000  MAIN_STARTUP Global Length=006E
CESXQQ . . . . . L WORD   0022  DATA    Global
CLNEQQ . . . . . L WORD   0002  DATA    Global          length
CRCXQQ . . . . . L WORD   001C  DATA    Global          of PROC
CRDXQQ . . . . . L WORD   001E  DATA    Global
CSXEQQ . . . . . L WORD   0000  DATA    Global
CURHQQ . . . . . L WORD   0014  DATA    Global
DOSOFF . . . . . L WORD   0020  DATA
DOSXQQ . . . . . F PROC   001E  ENTXCM  Global Length =0019
ENDHQQ . . . . . L WORD   0016  DATA    Global
ENDOQQ . . . . . L FAR    0000          External
ENDUQQ . . . . . L FAR    0000          External
ENDXQQ . . . . . L FAR    0005  ENTXCM  Global
ENDYQQ . . . . . L FAR    0000          External
ENTGQQ . . . . . L FAR    0000          External
FREXQQ . . . . . F PROC   006E  MAIN_STARTUP Global Length=0010
HDRFQQ . . . . . L WORD   0006  DATA    Global
HDRVQQ . . . . . L WORD   0008  DATA    Global
HEAPBEG. . . . . BYTE     0000  STACK  <----EQU statements
HEAPLOW. . . . . BYTE     0000  HEAP  <-    showing segment
INIUQQ . . . . . L FAR    0000          External
PNUXQQ . . . . . L WORD   0004  DATA    Global
RECEQQ . . . . . L WORD   0010  DATA    Global
REFEQQ . . . . . L WORD   000C  DATA    Global
REPEQQ . . . . . L WORD   000E  DATA    Global
RESEQQ . . . . . L WORD   000A  DATA    Global
SKTOP. . . . . . BYTE     0014  STACK  <-
SMLSTK . . . . . L NEAR   001C  MAIN_STARTUP
STARTMAIN. . . . F PROC   0000  ENTXCM Length=001E
STKBQQ . . . . . L WORD   0018  DATA   Global
STKHQQ . . . . . L WORD   001A  DATA    Global

        If Macro Assembler knows this length as one of the
        type lengths (BYTE, WORD, DWORD, QWORD,
        TBYTE), it shows that type name here.
```

This section lists all other symbolic values in your program that do not fit under the other categories.

Type shows the symbol's type:

L = Label
F = Far
N = Near
PROC = Procedure
Number
Alias |-----all defined by EQU or = directive
Text
Opcode

These entries may be combined to form the various types shown in the example.

For all procedures, the length of the procedure is given after its attribute (segment).

You may also see an entry under Type like:

```
L 0031
```

This entry results from code such as the following:

```
BAZ LABEL FOO
```

where FOO is a STRUC that is 31 bytes long.

BAZ will be shown in the symbol table with the L 0031 entry. Basically, Number (and some other similar entries) indicates that the symbol was defined by an EQU or = directive.

Value (usually) shows the numeric value the symbol represents. (In some cases, the Value column will show some text -- when the symbol was defined by EQU or = directive.)

Attr always shows the segment of the symbol, if known. Otherwise, the Attr column is blank. Following the segment name, the table will show either External, Global, or a blank (which means not declared with either the EXTRN or PUBLIC directive). The last entry applies to PROC types only. This is a length = entry, which is the length of the procedure.

If Type is Number, Opcode, Alias, or Text, the Symbols section of the listing will be structured differently. Whenever you see one of these four entries under Type, the symbol was created by an EQU directive or an = directive. All information that follows one of these entries is considered its "value," even if the "value" is simple text.

Each of the four types shows a value as follows:

Number shows a constant numeric value.

Opcode shows a blank. The symbol is an alias for an instruction mnemonic.

Sample directive statement: `FOO EQU ADD`

Alias shows a symbol name which the named symbol equals.

Sample directive statement: `FOO EQU BAX`

Text shows the "text" the symbol represents. "Text" is any other operand to an EQU directive that does not fit one of the other three categories above.

Sample directive statements:

```
GOO EQU 'WOW'
BAZ EQU DS:8[BX]
ZOO EQU 1.234
```

Contents

Chapter 6 **8087 Support**

6.1 Switches 6-1

CHAPTER 6

8087 SUPPORT

Macro Assembler supports standard Intel 8087 instructions and operands. A list of the instructions and opcodes can be found in Appendix C of this manual.

6.1 SWITCHES

There are two switches that are used when running Macro Assembler with an 8087. These switches are /R (for Real) and /E (for Emulate). The /R and /E switches are described below.

Switch	Function
/R	Use the /R switch when the code being produced by Macro Assembler is going to be run on a real 8087 machine (not an emulated machine). Code produced with the /R switch will only run on real 8087 machines.
/E	Use the /E switch when the code being produced by Macro Assembler is going to be run on an emulated 8087 machine. Code produced with the /E switch will also run on real 8087 machines with the appropriate emulator library.

The emulator library is provided with some MS-DOS language products. It contains specific 8087 emulation routines. Refer to your language compiler user's guide for information on the emulator library that has been provided. If your code is going to run on an emulated 8087 machine, you must specify the appropriate emulator library when you link your code with MS-LINK. If the library is not specified, MS-LINK will return errors for those unresolved symbols that are defined in the emulator library.

Contents

Chapter 7 Macro Assembler Messages

7.1 Operating Messages 7-1
7.2 Error Messages 7-2
  Assembler Errors 7-2
  I/O Handler Errors 7-13
  Runtime Errors 7-14
  Numerical List of Error Messages 7-15

CHAPTER 7

MACRO ASSEMBLER MESSAGES

Most of the messages output by Macro Assembler are error messages. The nonerror messages output by Macro Assembler are the banner Macro Assembler displays when first started, the command prompt messages, and the end of (successful) assembly message. These nonerror messages are classified here as operating messages. The error messages are classified as assembler errors, I/O handler errors, and runtime errors.

7.1 OPERATING MESSAGES

Banner Message and Command Prompts:

```
Macro Assembler v2.0 Copyright (C) Microsoft, Inc.

Source filename [.ASM]:
Object filename [source.OBJ]:
Source listing [NUL.LST]:
Cross reference [NUL.CRF]:
```

End of Assembly Message:

```
Warning   Fatal
Errors    Errors
n         n         (n=number of errors)
```

(your disk operating system's prompt)

7.2 ERROR MESSAGES

If the assembler encounters errors, error messages are output, along with the numbers of warning and fatal errors, and control is returned to your disk operating system. The message is output either to your terminal screen or to the listing file if you command one be created.

Error messages are divided into three categories: assembler errors, I/O handler errors, and runtime errors. In each category, messages are listed in alphabetical order with a short explanation where necessary. At the end of this chapter, the error messages are listed in a single numerical order list but without explanations.

Assembler Errors

Already defined locally (Code 23)

> Tried to define a symbol as EXTERNAL that had already been defined locally.

Already had ELSE clause (Code 7)

> Attempt to define an ELSE clause within an existing ELSE clause (you cannot nest ELSE without nesting IF...ENDIF).

Already have base register (Code 46)

> Trying to double base register.

Already have index register (Code 47)

> Trying to double index address

Block nesting error (Code 0)

> Nested procedures, segments, structures, macros, IRC, IRP, or REPT are not properly terminated. An example of this error is close of an outer level of nesting with inner level(s) still open.

Byte register is illegal (Code 58)

Use of one of the byte registers in context where it is illegal. For example, PUSH AL.

Can't override ES segment (Code 67)

Trying to override the ES segment in an instruction where this override is not legal. For example, store string.

Can't reach with segment reg (Code 68)

There is no ASSUME that makes the variable reachable.

Can't use EVEN on BYTE segment (Code 70)

Segment was declared to be byte segment and attempt to use EVEN was made.

Circular chain of EQU aliases (Code 83)

An alias EQU eventually points to itself.

Constant was expected (Code 42)

Expecting a constant and received something else.

CS register illegal usage (Code 59)

Trying to use the CS register illegally. For example, XCHG CS,AX.

Directive illegal in STRUC (Code 78)

All statements within STRUC blocks must either be comments preceded by a semicolon (;), or one of the Define directives.

Division by 0 or overflow (Code 29)

An expression is given that results in a divide by 0.

DUP is too large for linker (Code 74)

Nesting of DUP's was such that too large a record was created for the linker.

8087 opcode can't be emulated (Code 84)

Either the 8087 opcode or the operands you used with it produce an instruction that the emulator cannot support.

Extra characters on line (Code 1)

This occurs when sufficient information to define the instruction directive has been received on a line and superfluous characters beyond are received.

Field cannot be overridden (Code 80)

In a STRUC initialization statement, you tried to give a value to a field that cannot be overridden.

Forward needs override (Code 71)

This message is not currently used.

Forward reference is illegal (Code 17)

Attempt to forward reference something that must be defined in pass 1.

Illegal register value (Code 55)

The register value specified does not fit into the "reg" field (the reg field is greater than 7).

Illegal size for item (Code 57)

Size of referenced item is illegal. For example, shift of a double word.

Illegal use of external (Code 32)

Use of an external in some illegal manner. For example, DB M DUP(?) where M is declared external.

Illegal use of register (Code 49)

Use of a register with an instruction where there is no 8086 or 8088 instruction possible.

Illegal value for DUP count (Code 72)

DUP counts must be a constant that is not 0 or negative.

Improper operand type (Code 52)

Use of an operand such that the opcode cannot be generated.

Improper use of segment reg (Code 61)

Specification of a segment register where this is illegal. For example, an immediate move to a segment register.

Index displ. must be constant (Code 54)

Illegal use of index display.

Label can't have seg. override (Code 65)

Illegal use of segment override.

Left operand must have segment (Code 38)

Used something in right operand that required a segment in the left operand. (For example, ":.")

More values than defined with (Code 76)

Too many fields given in REC or STRUC allocation.

Must be associated with code (Code 45)

Use of data related item where code item was expected.

Must be associated with data (Code 44)

Use of code related item where data related item was exected. For example, MOV AX,<code-label>.

Must be AX or AL (Code 60)

Specification of some register other than AX or AL where only these are acceptable. For example, the IN instruction.

Must be index or base register (Code 48)

Instruction requires a base or index register and some other register was specified in square brackets, [ ].

Must be declared in pass 1 (Code 13)

Assembler expecting a constant value but got something else. An example of this might be a vector size being a forward reference.

Must be in segment block (Code 69)

Attempt to generate code when not in a segment.

Must be record field name (Code 33)

Expecting a record field name but got something else.

Must be record or field name (Code 34)

Expecting a record name or field name and received something else.

Must be register (Code 18)

Register unexpected as operand but you furnished a symbol -- was not a register.

Must be segment or group (Code 20)

Expecting segment or group and something else was specified.

Must be structure field name (Code 37)

Expecting a structure field name but received something else.

Must be symbol type (Code 22)

Must be WORD, DW, QW, BYTE, or TB but received something else.

Must be var, label or constant (Code 36)

Expecting a variable, label, or constant but received something else.

Must have opcode after prefix (Code 66)

Use of one of the prefix instructions without specifying any opcode after it.

Near JMP/CALL to different CS (Code 64)

Attempt to do a NEAR jump or call to a location in a different CS ASSUME.

No immediate mode (Code 56)

Immediate mode specified or an opcode that cannot accept the immediate. For example, PUSH.

No or unreachable CS (Code 62)

Trying to jump to a label that is unreachable.

Normal type operand expected (Code 41)

Received STRUCT, FIELDS, NAMES, BYTE, WORD, or DW when expecting a variable label.

Not in conditional block (Code 8)

An ENDIF or ELSE is specified without a previous conditional assembly directive active.

Not proper align/combine type (Code 25)

SEGMENT parameters are incorrect.

One operand must be const (Code 39)

This is an illegal use of the addition operator.

Only initialize list legal (Code 77)

Attempt to use STRUC name without angle brackets, < >.

Operand combination illegal (Code 63)

Specification of a two-operand instrucion where the combination specified is illegal.

Operands must be same or 1 abs (Code 40)

Illegal use of the subtraction operator.

Operand must have segment (Code 43)

Illegal use of SEG directive.

Operand must have size (Code 35)

Expected operand to have a size, but it did not.

Operand not in IP segment (Code 51)

Access of operand is impossible because it is not in the current IP segment.

Operand types must match (Code 31)

Assembler gets different kinds or sizes of arguments in a case where they must match. For example, MOV.

Operand was expected (Code 27)

Assembler is expecting an operand but an operator was received.

Operator was expected (Code 28)

Assembler was expecting an operator but an operand was received.

Override is of wrong type (Code 81)

In a STRUC initialization statement, you tried to use the wrong size on override. For example, 'HELLO' for DW field.

Override with DUP is illegal (Code 79)

In a STRUC initialization statement, you tried to use DUP in an override.

Phase error between passes (Code 6)

The program has ambiguous instruction directives such that the location of a label in the program changed in value between pass 1 and pass 2 of the assembler. An example of this is a forward reference coded without a segment override where one is required. There would be an additional byte (the code segment override) generated in pass 2 causing the next label to change. You can use the /D switch to produce a listing to aid in resolving phase errors between passes (see Section 5.4, "Macro Assembler Command Switches").

Redefinition of symbol (Code 4)

This error occurs on pass 2 and succeeding definitions of a symbol.

Reference to mult defined (Code 26)

The instruction references something that has been multi-defined.

Register already defined (Code 2)

This will only occur if the assembler has internal logic errors.

Register can't be forward ref (Code 82)

Relative jump out of range (Code 53)

> Relative jumps must be within the range -128 +127 of the current instruction, and the specific jump is beyond this range.

Segment parameters are changed (Code 24)

> List of arguments to SEGMENT were not identical to the first time this segment was used.

Shift count is negative (Code 30)

> A shift expression is generated that results in a negative shift count.

Should have been group name (Code 12)

> Expecting a group name but something other than this was given.

Symbol already different kind (Code 15)

> Attempt to define a symbol differently from a previous definition.

Symbol already external (Code 73)

> Attempt to define a symbol as local that is already external.

Symbol has no segment (Code 21)

> Trying to use a variable with SEG, and the variable has no known segment.

Symbol is multi-defined (Code 5)

> This error occurs on a symbol that is later redefined.

Symbol is reserved word (Code 16)

> Attempt to use an assembler reserved word illegally. (For example, to declare MOV as a variable.)

Symbol not defined (Code 9)

A symbol is used that has no definition.

Symbol type usage illegal (Code 14)

Illegal use of a PUBLIC symbol.

Syntax error (Code 10)

The syntax of the statement does not match any recognizable syntax.

Type illegal in context (Code 11)

The type specified is of an unacceptable size.

Unknown symbol type (Code 3)

Symbol statement has something in the type field that is unrecognizable.

Usage of ? (indeterminate) bad (Code 75)

Improper use of the "?". For example, ?+5.

Value is out of range (Code 50)

Value is too large for expected use. For example, MOV AL,5000.

Wrong type of register (Code 19)

Directive or instruction expected one type of register, but another was specified. For example, INC CS.

I/O Handler Errors

These error messages are generated by the I/O handlers. These messages appear in a different format from the Assembler Errors:

```
MASM Error -- error-message-text
  in:  filename
```

The filename is the name of the file being handled when the error occurred.

The error-message-text is one of the following messages:

Data format (Code 114)

Device full (Code 108)

Device name (Code 102)

Device offline (Code 105)

File in use (Code 112)

File name (Code 107)

File not found (Code 110)

File not open (Code 113)

File system (Code 104)

Hard data (Code 101)

Line too long (Code 115)

Lost file (Code 106)

Operation (Code 103)

Protected file (Code 111)

Unknown device (Code 109)

Runtime Errors

These messages may be displayed as your assembled program is being executed.

Internal Error

> Usually caused by an arithmetic check. If it occurs, notify Microsoft Corporation.

Out of Memory

> This message has no corresponding number. Either the source was too big or too many labels are in the symbol table.

Numerical Order List of Error Messages

Code	Message
0	Block nesting error
1	Extra characters on line
2	Register already defined
3	Unknown symbol type
4	Redefinition of symbol
5	Symbol is multi-defined
6	Phase error between passes
7	Already had ELSE clause
8	Not in conditional block
9	Symbol not defined
10	Syntax error
11	Type illegal in context
12	Should have been group name
13	Must be declared in pass 1
14	Symbol type usage illegal
15	Symbol already different kind
16	Symbol is reserved word
17	Forward reference is illegal
18	Must be register
19	Wrong type of register
20	Must be segment or group
21	Symbol has no segment
22	Must be symbol type
23	Already defined locally
24	Segment parameters are changed
25	Not proper align/combine type
26	Reference to mult defined
27	Operand was expected
28	Operator was expected
29	Division by 0 or overflow
30	Shift count is negative
31	Operand types must match
32	Illegal use of external
33	Must be record field name
34	Must be record or field name
35	Operand must have size
36	Must be var, label or constant
37	Must be structure field name
38	Left operand must have segment
39	One operand must be const
40	Operands must be same or 1 abs
41	Normal type operand expected
42	Constant was expected
43	Operand must have segment
44	Must be associated with data
45	Must be associated with code
46	Already have base register
47	Already have index register
48	Must be index or base register
49	Illegal use of register
50	Value is out of range

51 Operand not in IP segment
52 Improper operand type
53 Relative jump out of range
54 Index displ. must be constant
55 Illegal register value
56 No immediate mode
57 Illegal size for item
58 Byte register is illegal
59 CS register illegal usage
60 Must be AX or AL
61 Improper use of segment reg
62 No or unreachable CS
63 Operand combination illegal
64 Near JMP/CALL to different CS
65 Label can't have seg. override
66 Must have opcode after prefix
67 Can't override ES segment
68 Can't reach with segment reg
69 Must be in segment block
70 Can't use EVEN on BYTE segment
71 Forward needs override
72 Illegal value for DUP count
73 Symbol already external
74 DUP is too large for linker
75 Usage of ? (indeterminate) bad (Code 75)
76 More values than defined with
77 Only initialize list legal
78 Directive illegal in STRUC
79 Override with DUP is illegal
80 Field cannot be overridden
81 Override is of wrong type
82 Register can't be forward ref
83 Circular chain of EQU aliases
84 8087 opcode can't be emulated

101	Hard data
102	Device name
103	Operation
104	File system
105	Device offline
106	Lost file
107	File name
108	Device full
109	Unknown device
110	File not found
111	Protected file
112	File in use
113	File not open
114	Data format
115	Line too long

Contents

Appendix A ASCII Character Codes

Appendix B Table of Macro Assembler Directives

B.1 Memory Directives B-1
B.2 Macro Directives B-2
B.3 Conditional Directives B-2
B.4 Listing Directives B-2
B.5 Attribute Operators B-3
B.6 Precedence of Operators B-4

Appendix C Table of 8086 and 8087 Instructions

C.1 8086 Instruction Mnemonics, Alphabetical C-1
C.2 8087 Instruction Mnemonics, Alphabetical C-4
C.3 8086 Instruction Mnemonics by Argument Type C-6
C.4 8087 Instruction Mnemonics by Argument Type C-10

APPENDIX A

ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	033	21H	!
001	01H	SOH	034	22H	"
002	02H	STX	035	23H	£ = #
003	03H	ETX	036	24H	$
004	04H	EOT	037	25H	%
005	05H	ENQ	038	26H	&
006	06H	ACK	039	27H	,
007	07H	BEL	040	28H	(
008	08H	BS	041	29H	)
009	09H	HT	042	2AH	*
010	0AH	LF	043	2BH	+
011	0BH	VT	044	2CH	,
012	0CH	FF	045	2DH	-
013	0DH	CR	046	2EH	.
014	0EH	SO	047	2FH	/
015	0FH	SI	048	30H	0
016	10H	DLE	049	31H	1
017	11H	DC1	050	32H	2
018	12H	DC2	051	33H	3
019	13H	DC3	052	34H	4
020	14H	DC4	053	35H	5
021	15H	NAK	054	36H	6
022	16H	SYN	055	37H	7
023	17H	ETB	056	38H	8
024	18H	CAN	057	39H	9
025	19H	EM	058	3AH	:
026	1AH	SUB	059	3BH	;
027	1BH	ESCAPE	060	3CH	<
028	1CH	FS	061	3DH	=
029	1DH	GS	062	3EH	>
030	1EH	RS	063	3FH	?
031	1FH	US	064	40H	~~#~~ @
032	20H	SPACE			

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

Dec	Hex	CHR	Dec	Hex	CHR
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	]	123	7BH	}
092	5CH	\	124	7CH	|
093	5DH	[	125	7DH	{
094	5EH	^	126	7EH	@
095	5FH		128	7FH	DEL
096	60H	⊤			

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX B

TABLE OF MACRO ASSEMBLER DIRECTIVES

B.1 MEMORY DIRECTIVES

```
             ASSUME <seg-reg>:<seg-name>],<seg-reg>:
                     <seg-name>...[
             ASSUME NOTHING
             COMMENT <delim><text><delim>

      <name> DB <exp>
      <name> DD <exp>
      <name> DQ <exp>
      <name> DT <exp>
      <name> DW <exp>

             END ]<exp>[
      <name> EQU <exp>
      <name> = <exp>
             EXTRN <name>:<type>],<name>:<type>...[
             PUBLIC <name>],<name>...[
      <name> LABEL <type>
             NAME <module-name>

      <name> PROC ]NEAR[
      <name> PROC ]FAR[
               |
 <proc-name> ENDP

             .RADIX <exp>
      <name> RECORD <field>:<width>]=<exp>[],...[

      <name> GROUP <segment-name>],...[
      <name> SEGMENT ]<align>[]<combine>[]<class>[
               |
  <seg-name> ENDS
             EVEN
             ORG <exp>

      <name> STRUC
               |
<struc-name> ENDS
```

B.2 MACRO DIRECTIVES

```
         ENDM
         EXITM
         IRP <dummy>,<parameters in angle brackets>
         IRPC <dummy>,string
         LOCAL <parameter>],<parameter>...[
  <name> MACRO <parameter>],<parameter>...[
         PURGE <macro-name>],...[
         REPT <exp>
```

Special Macro Operators
& (ampersand) - concantenation
<text> (angle brackets - single literal)
;; (double semicolons) - suppress comment
! (exclamation point) - next character literal
% (percent sign) - convert expression to number

B.3 CONDITIONAL DIRECTIVES

```
ELSE
IF <exp>
IFB <arg>
IFDEF <symbol>
IFDIF <arg1>,<arg2>
IFE <exp>
IFIDN <arg1>,<arg2>
IFNB <arg>
IFNDEF <symbol>
IF1
IF2
```

B.4 LISTING DIRECTIVES

```
.CREF
.LALL
.LFCOND
.LIST
%OUT <text>
PAGE <exp>
.SALL
.SFCOND
SUBTTL <text>
.TFCOND
TITLE <text>
.XALL
.XCREF
.XLIST
```

B.5 ATTRIBUTE OPERATORS

Override operators

```
Pointer (PTR)
    <attribute>   PTR   <expression>
Segment Override (:) (colon)
    <segment-register>:<address-expression>
    <segment-name>:<address-expression>
    <group-name>:<address-expression>
SHORT
    SHORT <label>
THIS
    THIS <distance>
    THIS <type>
```

Value Returning Operators

```
SEG
    SEG <label>
    SEG <variable>
OFFSET
    OFFSET <label>
    OFFSET <variable>
TYPE
    TYPE <label>
    TYPE <variable>
.TYPE
    .TYPE <variable>
LENGTH
    LENGTH <variable>
SIZE
    SIZE <variable>
```

Record Specific operators

```
Shift-count  - (Record fieldname)
    <record-fieldname>
MASK
    MASK <record-fieldname>
WIDTH
    WIDTH <record-fieldname>
    WIDTH <record>
```

B.6 PRECEDENCE OF OPERATORS

All operators in a single item have the same precedence, regardless of the order listed within the item. Spacing and line breaks are used for visual clarity, not to indicate functional relations.

1. LENGTH, SIZE, WIDTH, MASK
   Entries inside: parenthesis ( )
   angle brackets < >
   square brackets] [
   structure variable operand: <variable>.<field>

2. segment override operator: colon (:)

3. PTR, OFFSET, SEG, TYPE, THIS

4. HIGH, LOW

5. *, /, MOD, SHL, SHR

6. +, - (both unary and binary)

7. EQ, NE, LT, LE, GT, GE

8. Logical NOT

9. Logical AND

10. Logical OR, XOR

11. SHORT,.TYPE

APPENDIX C

TABLE OF 8086 AND 8087 INSTRUCTIONS

Macro Assembler supports both the 8086 and 8087 mnemonics. The mnemonics are listed alphabetically with their full names. The 8086 instructions are also listed in groups based on the type of arguments the instruction takes.

C.1 8086 INSTRUCTION MNEMONICS, ALPHABETICAL

Mnemonic	Full Name
AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiplication
AAS	ASCII adjust for subtraction
ADC	Add with carry
ADD	Add
AND	AND
CALL	CALL
CBW	Convert byte to word
CLC	Clear carry flag
CLD	Clear direction flag
CLI	Clear interrupt flag
CMC	Complement carry flag
CMP	Compare
CMPS	Compare byte or word (of string)
CMPSB	Compare byte string
CMPSW	Compare word string
CWD	Convert word to double word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC	Decrement
DIV	Divide
ESC	Escape
HLT	Halt
IDIV	Integer divide
IMUL	Integer multiply
IN	Input byte or word
INC	Increment
INT	Interrupt
INTO	Interrupt on overflow

IRET	Interrupt return
JA	Jump on above
JAE	Jump on above or equal
JB	Jump on below
JBE	Jump on below or equal
JC	Jump on carry
JCXZ	Jump on CX zero
JE	Jump on equal
JG	Jump on greater
JGE	Jump on greater or equal
JL	Jump on less than
JLE	Jump on less than or equal
JMP	Jump
JNA	Jump on not above
JNAE	Jump on not above or equal
JNB	Jump on not below
JNBE	Jump on not below or equal
JNC	Jump on no carry
JNE	Jump on not equal
JNG	Jump on not greater
JNGE	Jump on not greater or equal
JNL	Jump on not less than
JNLE	Jump on not less than or equal
JNO	Jump on not overflow
JNP	Jump on not parity
JNS	Jump on not sign
JNZ	Jump on not zero
JO	Jump on overflow
JP	Jump on parity
JPE	Jump on parity even
JPO	Jump on parity odd
JS	Jump on sign
JZ	Jump on zero
LAHF	Load AH with flags
LDS	Load pointer into DS
LEA	Load effective address
LES	Load pointer into ES
LOCK	LOCK bus
LODS	Load byte or word (of string)
LODSB	Load byte (string)
LODSW	Load word (string)
LOOP	LOOP
LOOPE	LOOP while equal
LOOPNE	LOOP while not equal
LOOPNZ	LOOP while not zero
LOOPZ	LOOP while zero
MOV	Move
MOVS	Move byte or word (of string)
MOVBS	Move byte (string)
MOVSW	Move word (string)
MUL	Multiply
NEG	Negate
NOP	No operation
NOT	NOT
OR	OR

OUT	Output byte or word
POP	POP
POPF	POP flags
PUSH	PUSH
PUSHF	PUSH flags
RCL	Rotate through carry left
RCR	Rotate through carry right
REP	Repeat
RET	Return
ROL	Rotate left
ROR	Rotate right
SAHF	Store AH into flags
SAL	Shift arithmetic left
SAR	Shift arithmetic right
SBB	Subtract with borrow
SCAS	Scan byte or word (of string)
SCASB	Scan byte (string)
SCASW	Scan word (string)
SHL	Shift left
SHR	Shift right
STC	Set carry flag
STD	Set direction flag
STI	Set interrupt flag
STOS	Store byte or word (of string)
STOSB	Store byte (string)
STOSW	Store word (string)
SUB	Subtract
TEST	TEST
WAIT	WAIT
XCHG	Exchange
XLAT	Translate
XOR	Exclusive OR

C.2 8087 INSTRUCTION MNEMONICS, ALPHABETICAL

Mnemonic	Full Name
F2XM1	Calculate 2X-1
FABS	Take absolute value of top of stack
FADD	Add real
FADDP	Add real and pop stack
FBLD	Load packed decimal onto top of stack
FBSTP	Store packed decimal and pop stack
FCHS	Change sign on the top stack element
FCLEX	Clear exceptions after WAIT
FCOM	Compare real
FCOMP	Compare real and pop stack
FCOMPP	Compare real and pop stack twice
FDECSTP	Decrement stack pointer
FDISI	Disable interrupts after WAIT
FDIV	Divide real
FDIVP	Divide real and Pop stack
FDIVR	Reversed real divide
FDIVRP	Reversed real divide and pop stack twice
FENI	Enable interrupts after WAIT
FFREE	Free stack element
FIADD	Add integer
FICOM	Integer compare
FICOMP	Integer compare and pop stack
FIDIV	Integer divide
FIDIVR	Reversed integer divide
FILD	Load integer onto top of stack
FIMUL	Integer multiply
FINCSTP	Increment stack pointer
FINIT	Initialize processor after WAIT
FIST	Store integer
FISTP	Store integer and pop stack
FISUB	Integer subtract
FISUBR	Reversed integer subtract
FLD	Load real onto top of stack
FLD1	Load +1.0 onto top of stack
FLDCW	Load control word
FLDENV	Load 8087 environment
FLDL2E	Load log 2 e onto top of stack
FLDL2T	Load log 2 10 onto top of stack
FLDLG2	Load log 10 2 onto top of stack
FLDLN2	Load log e 2 onto top of stack
FLDPI	Load pi onto top of stack
FLDZ	Load +0.0 onto top of stack

FMUL	Multiply real
FMULP	Multiply real and pop stack
FNCLEX	Clear exceptions with no WAIT
FNDISI	Disable interrupts with no WAIT
FNENI	Enable interrupts with no WAIT
FNINIT	Initialize processor, with no WAIT
FNOP	No operation
FNSAVE	Save 8087 state with no WAIT
FNSTCW	Store control word without WAIT
FNSTENV	Store 8087 environment with no WAIT
FNSTSW	Store 8087 status word with on WAIT
FPATAN	Partial arctangent function
FPREM	Partial remainder
FPTAN	Partial tangent function
FRNDINT	Round to integer
FRSTOR	Restore state
FSAVE	Save 8087 state after WAIT
FSCALE	Scale
FSQRT	Square root
FST	Store real
FSTCW	Store control word with WAIT
FSTENV	Store 8087 environment after WAIT
FSTP	Store real and pop stack
FSTSW	Store 8087 status word after WAIT
FSUB	Subtract real
FSUBP	Subtract real and pop stack
FSUBR	Reversed real subtract
FSUBRP	Reversed real subtract and pop stack
FTST	Test top of stack
FWAIT	Wait for last 8087 operation to complete
FXAM	Examine top of stack element
FXCH	Exchange contents of stack element and stack top
FXTRACT	Extract exponent and significand from number in top of stack
FYL2X	Calculate Y:log 2 X
FYL2PI	Calculate Y:log 2 (x+1)

C.3 8086 INSTRUCTION MNEMONICS BY ARGUMENT TYPE

In this section, the instructions are grouped according to the type of argument(s) they take. In each group the instructions are listed alphabetically in the first column. The formats of the instructions with the valid argument types are shown in the second column. If a format shows OP, that format is legal for all the instructions shown in that group. If a format is specific to one mnemonic, the mnemonic is shown in the format instead of OP.

The following abbreviations are used in these lists:

OP = opcode; instruction mnemonic

reg = byte register (AL,AH,BL,BH,CL,CH,DL,DH)
or word register (AX,BX,CX,DX,SI,DI,BP,SP)

r/m = register or memory address or indexed and/or based

accum = AX or AL register

immed = immediate

mem = memory operand

segreg = segment register (CS,DS,SS,ES)

General 2 operand instructions

Mnemonics	Argument Types
ADC	OP reg,r/m
ADD	OP r/m,reg
AND	OP accum,immed
CMP	OP r/m,immed
OR	
SBB	
SUB	
TEST	
XOR	

In addition, add to the arguments a sign extent for word immediate.

CALL and JUMP type instructions

Mnemonics	Argument Types
CALL	OP mem }NEAR{}FAR{ direction
JMP	OP r/m (indirect data -- DWORD, WORD)

Relative jumps

Argument Type

OP addr (+129 or -126 of IP at start, or
+127 at end of jump instruction)

Mnemonics

JA
JNBE
JAE
JNB
JNC
JB
JC
JNAE
JBE
JNA
JCXZ
JE
JZ
JG
JNLE
JGE
JNL
JL
JNGE
JLE
JNG
JNE
JNZ
JNO
JNP
JPO
JNS
JO
JP
JPE
JS

Loop instructions : same as Relative jumps

LOOP
LOOPE
LOOPZ
LOOPNE
LOOPNZ

Return instruction

Mnemonic	Argument Type
RET	]immed[(optional, number of words to POP)

No operand instructions

Mnemonics

AAA
AAD
AAM
AAS
CBW
CLC
CLD
CLI
CMC
CMPSB
CMPSW
CWD
DAA
DAS
HLT
INTO
IRET
LAHF
LODSB
LODSW
MOVSB
MOVSW
NOP
POPF
PUSHF
SAHF
SCASB
SCASW
STC
STD
STI
STOSB
STOSW
WAIT
XLATB

Load instructions

Mnemonics	Argument Type
LDS LEA LES	OP r/m (except that OP reg is illegal)

Move instructions

Mnemonic	Argument Types
MOV	OP mem,accum OP accum,mem OP segreg,r/m (except CS is illegal) OP r/m,segreg OP r/m,reg OP reg,r/m OP reg,immed OP r/m,immed

Push and pop instructions

Mnemonics	Argument Types
PUSH POP	OP word-reg OP segreg (POP CS is illegal) OP r/m

Shift/rotate type instructions

Mnemonics	Argument Types
RCL RCR ROL ROR SAL SHL SAR SHR	OP r/m,1 OP r/m,CL

Input/output instructions

Mnemonics	Argument Types
IN	IN accum,byte-immed (immed = port 0-255) IN accum,DX
OUT	OUT immed,accum OUT DX,accum

Increment/decrement instructions

Mnemonics	Argument Types
INC	OP word-reg
DEC	OP r/m

Arith. multiply/division/negate/not

Mnemonics	Argument Type
DIV	OP r/m (implies AX OP r/m, except NEG)
IDIV	
MUL	
IMUL	
NEG	(NEG implies AX OP NOP)
NOT	

Interrupt instruction

Mnemonic	Argument Types
INT	INT 3 (value 3 is one-byte instruction) INT byte-immed

Exchange instruction

Mnemonic	Argument Types
XCHG	XCHG accum,reg XCHG reg,accum XCHG reg,r/m XCHG r/m,reg

Miscellaneous instructions

Mnemonics	Argument Types
XLAT	XLAT byte-mem (only checks argument, not in opcode)
ESC	ESC 6-bit-number,r/m

String primitives

These instructions have bits to record only their operand(s), if they are byte or word, and if a segment override is involved.

Mnemonics	Argument Types
CMPS	CMPS byte-word,byte-word (CMPS right operand is ES)
LODS	LODS byte/word,byte/word (LODS one argument = no ES)
MOVS	MOVS byte/word,byte/word (MOVS left operand is ES)
SCAS	SCAS byte/word,byte/word (SCAS one argument = ES)
STOS	STOS byte/word,byte/word (STOS one argument = ES)

Repeat prefix to string instructions

Mnemonics

LOCK
REP
REPE
REPZ
REPNE
REPNZ

C.4 8087 INSTRUCTION MNEMONICS BY ARGUMENT TYPE

No operands

F2XM1
FABS
FCHS
FCLEX
FCOMPP
FDECSTP
FDISI
FENI
FINCSTP
FINIT
FLD1
FLD2E
FLD2T
FLDLG2
FLDLN2
FLDPI
FLDZ
FNCLEX
FNDISI
FNENI
FNINIT
FNOP
FPATAN
FPREM
FPTAN
FRNDINT
FSCALE
FSQRT
FTST
FXAM
FXTRACT
FYL2X
FYL2XP1
FWAIT

2-Argument Floating Arithmatic

Mnemonics	Argument Types
FADD	Blank
FDIV	mem 4,8 bytes
FDIVR	ST,ST(i)
FMUL	ST(i),ST
FSUB	
FSUBR	

Stack only floating point arithmatic

Mnemonics	Argument Types
FADDP	ST(i)
FDIVP	ST
FDIVRP	
FMULP	
FSUBP	
FSUBRP	

Compare and store using stack

Mnemonics	Argument Types
FCOM	ST
FCOMP	ST(i)
FST	blank

Stack

Mnemonics	Argument Types
FFREE	ST(i)
FXCH	blank

Integer arithmatic

Mnemonics	Argument Types
FIADD	mem 2,4 bytes
FICOM	
FICOMP	
FIDIV	
FIDIVR	
FIMUL	
FIST	
FISUB	
FISUBR	

Floating point load/store memory

Mnemonics	Argument Types
FLD FSTP	mem 4,8, or 10 bytes

Integer load/store memory

Mnemonics	Argument Types
FILD FISTP	mem 2,4, or 8 bytes

Load/store control or status

Mnemonics	Argument Types
FLDCW FNSTCW FNSTSW FSTCW FSTSW	mem 2 bytes

Save/Restore 8087 environment

Mnemonics	Argument Types
FLDENV FNSTENV FSTENV	mem 14 bytes

94-byte memory (8087 Save/Restore entire state)

Mnemonics	Argument Types
FNSAVE FRSTOR FSAVE	mem 94 bytes

BCD load/store

Mnemonics	Argument Types
FBLD FBSTP	mem 10 bytes

INDEX

! 4-53
% 4-56
%OUT 4-60
& 4-53
/D (assembler switch) 5-9
/D (MACRO-86) 7-9
/O (assembler switch) 5-9
/X (assembler switch) 5-9
8087 support 6-1
: (colon - segment override operator) 3-19
; (command character) 5-4
;; 4-53
<record-fieldname
  (shift count) 3-30
= (equal sign directive) . . . 4-12

Action 1-9, 4-1
Arithmetic operators 3-33
Assembler errors 7-2

Calling a Macro 4-44
Colon (: - segment override operator) 3-19
Command Characters 5-4
  ; 5-4
  CONTROL-C 5-4
Command Prompts
  Cross-reference 5-2
  Object filename 5-2
  Source filename 5-2
  Source listing 5-2
  Summary of 5-2
COMMENT 4-6
Comments 1-8
CONTROL-C (command character) 5-4

Data items 3-9
DB - Define Byte . . . . 4-7 to 4-8
DD - Define Doubleword 4-9

DD - Define Doubleword 4-7
Direct memory operands 3-13
Directives 4-1, 4-3
  %OUT 4-60
  .CREF 4-63
  .LALL 4-62
  .LFCOND 4-62
  .LIST 4-61
  .RADIX 4-26
  .SALL 4-62
  .SFCOND 4-62
  .TFCOND 4-62
  .XALL 4-62
  .XCREF 4-63
  .XLIST 4-61
  = (equal sign) 4-12
  COMMENT 4-6
  Conditional 4-37
  DB - Define Byte . . . . 4-7 to 4-8
  DD - Define Doubleword . . . 4-9
  DQ - Define Quadword 4-9
  DT - Define Tenbytes 4-7, 4-9
  DW - Define Word 4-8
  ELSE 4-40
  END 4-10
  ENDC 4-40
  ENDIF 4-40
  ENDM 4-45
  ENDP 4-23
  EQU 4-11
  EVEN 4-13
  EXITM 4-46
  EXTRN 4-14
  GROUP 4-16
  IF 4-38
  IF1 4-38
  IF2 4-38
  IFB 4-39
  IFDEF 4-38
  IFDIF 4-40
  IFE 4-38
  IFIDN 4-40
  IFNB 4-39
  IFNDEF 4-38
  INCLUDE 4-18
  IRP 4-51
  IRPC 4-52
  LABEL 4-19
  Listing 4-57
  LOCAL 4-47
  MACRO 4-42
  Memory 4-5
  NAME 4-21
  PAGE 4-57
  PROC 4-23

  PUBLIC 4-25
  PURGE 4-48
  RECORD 4-27
  REPT 4-50
  SEGMENT 4-30
  STRUCTURE 4-34
  SUBTTL 4-59
  TITLE 4-21, 4-58
Directives
  DD - Define Doubleword . . . 4-7
  DQ - Define Quadword 4-7
  DW - Define Word 4-7
DQ - Define Quadword 4-9
DQ - Define Quadword 4-7
DT - Define Tenbytes 4-7, 4-9
DW - Define Word 4-8
DW - Define Word 4-7

ELSE 4-40
END 4-10
ENDC 4-40
ENDIF 4-40
ENDM 4-45
ENDP 4-23
ENDS 4-34
EQU 4-11
Equal sign directive (=) . . . 4-12
Error messages
  numerical list 7-14
EVEN 4-13
EXITM 4-46
Expression evaluation 3-36
Expressions 1-10, 3-1
EXTRN 4-14

Formats
  program listing 5-10
  symbol table 5-17
Formats of listings and symbol tables 5-10

General Facts about Source Files 1-1
GROUP 4-16

HIGH 3-22

I/O Handler errors 7-12
IF 4-38
IF1 4-38
IF2 4-38
IFB 4-39
IFDEF 4-38
IFDIF 4-40
IFE 4-38
IFIDN 4-40
IFNB 4-39

IFNDEF 4-38
Immediate operands 3-9
INCLUDE 4-18
Indexed memory operands . . . 3-14
Instructions . . . . 4-1 to 4-2
Instructions by argument type (Appendices) C-6
Instructions, alphabetical (Appendices) C-1
IRP 4-51
IRPC 4-52

LABEL 4-19
Labels 2-2
Legal characters 1-2
LENGTH 3-27
LOCAL 4-47
Logical operators 3-35
LOW 3-22

MACRO 4-42
MASK 3-32
Memory directives 4-5
Memory operands 3-13
Memory organization 3-2

NAME 4-21
Names 1-6, 2-1
Numeric notation 1-3, 3-9

OFFSET 3-24
Offset attribute 2-3
Operand summary 1-12
Operands 3-8
Operator summary 1-12
Operators 3-17
ORG 4-22
Override operators 3-18
Overviews
  MACRO-86 operation 8

PAGE 4-57
Pass 1 listing versus pass 2 listing 5-16
Pointer (PTR) 3-18
Precedence of operators . . . 3-36
PROC 4-23
Program listing format 5-10
PTR 3-18
PUBLIC 4-25
PURGE 4-48

RECORD 4-27
Register operands 3-10
Relational operators 3-34
REPT 4-50
Runtime errors 7-13

SEG 3-23
SEGMENT 4-30
Segment attribute 2-3
Segment override operator (:) 3-19
Shift count 3-30
SHORT 3-20
SIZE 3-28
Source file contents 1-4
Source file naming 1-1
Special Macro Operators . . . 4-53
  ! 4-53
  % 4-56
  & 4-53
  ;; 4-53
Statement Format
  Action 1-9
  Comments 1-8
  Directives 1-5
  Expressions 1-10
  Instructions 1-5
  Names 1-6
Statement line format 1-5
STRUCTURE 4-34
Structure operands 3-15
SUBTTL 4-59
Summary
  Operands 1-12
  Operators 1-12
Summary of listing symbols . . 5-12
Summary of methods to invoke MACRO-86 5-1
Switches
  MACRO-86
    Summary of 5-9
  MACRO-86
    /D 5-9, 7-9
    /O 5-9
    /X 5-9
Switches (MACRO-86) 5-7
Symbol table format 5-17
Symbols 2-7

THIS 3-21
TITLE 4-21, 4-58
TYPE 3-25
Type attribute 2-4, 2-6

Value returning operators . . 3-23
Variables 2-5

WIDTH 3-31

SEATTLE COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® LINK

Linker Utility

for 8086 and 8088 Microprocessors

LINK

System Requirements

The Microsoft Linker Utility requires:

50K bytes of memory minimum:
- 40K bytes for code and data
- 10K bytes for run space

Disk drive(s):

1 disk drive if and only if output is sent to the same physical disk from which the input was taken. MS-LINK does not allow time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® LINK

Linker Utility

for 8086 and 8088 Microprocessors

ADDENDUM to the Microsoft MS-DOS **Macro Assembler Manual**

MS-LINK

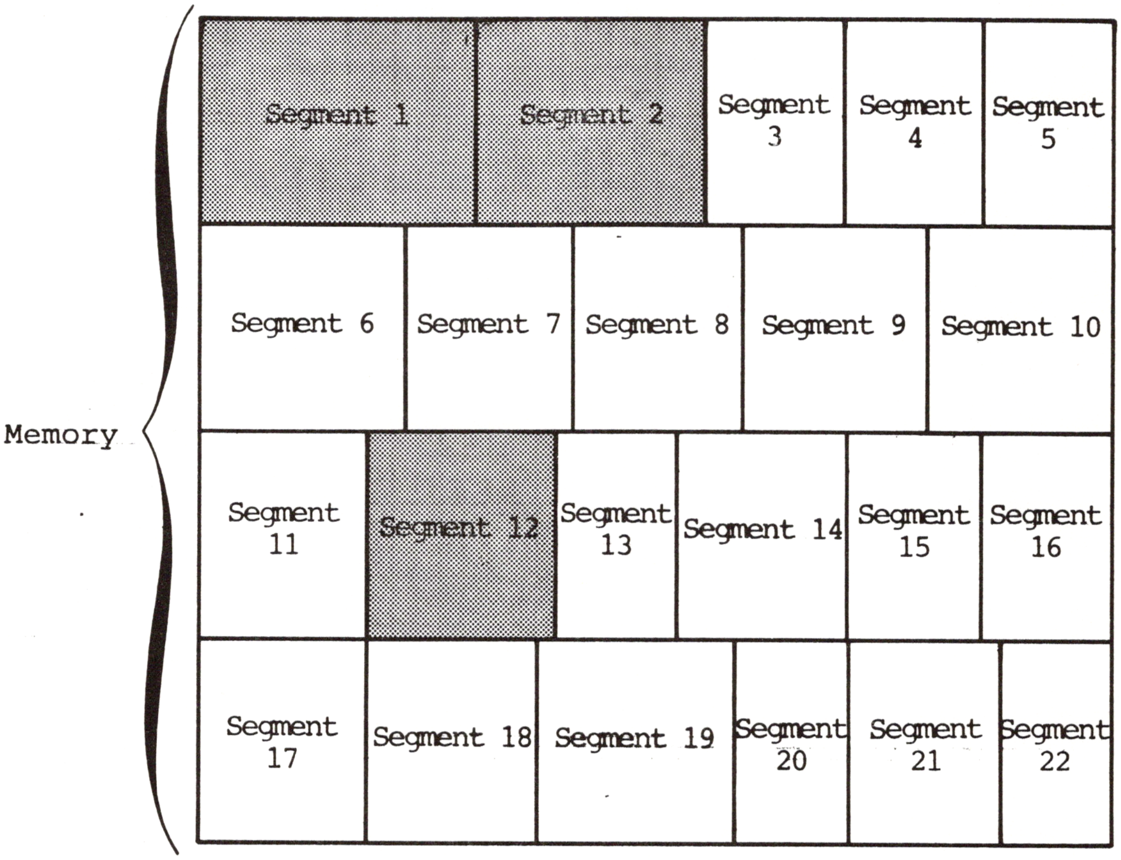
NOTE

References in the Macro Assembler Manual to the MS-DOS User's Guide refer to this addendum.

1.0 DEFINITIONS

Some of the terms used in the MS-LINK section of this manual are explained below to help you understand how MS-LINK works. Generally, if you are linking object modules compiled from BASIC, Pascal, or a high-level language, you will not need to know these terms. If you are writing and compiling programs in assembly language, however, you will need to understand MS-LINK and the definitions described below.

In MS-DOS, memory can be divided into segments, classes, and groups. Figure 1 illustrates these concepts.



Memory
Segment 1
Segment 2
Segment 3
Segment 4
Segment 5
Segment 6
Segment 7
Segment 8
Segment 9
Segment 10
Segment 11
Segment 12
Segment 13
Segment 14
Segment 15
Segment 16
Segment 17
Segment 18
Segment 19
Segment 20
Segment 21
Segment 22

shaded area = a group (64K bytes addressable)

Figure 1. How Memory Is Divided

Example:

	Segment Name	Segment Class Name
Segment 1	PROG.1	CODE
Segment 2	PROG.2	CODE
Segment 12	PROG.3	DATA

Note that segments 1, 2, and 12 have different segment names but may or may not have the same segment class name. Segments 1, 2, and 12 form a group, with a group address of the lowest address of segment 1 (i.e., the lowest address in memory).

Each segment has a segment name and a class name. MS-LINK loads all segments into memory by class name, from the first segment encountered to the last. All segments assigned to the same class are loaded into memory continguously.

During processing, MS-LINK references segments by their addresses in memory (where they are located). MS-LINK does this by finding groups of segments.

A group is a collection of segments that fit within a 64K byte area of memory. The segments do not need to be contiguous to form a group (see Figure 1). The address of any group is the lowest address of the segments in that group. At link time, MS-LINK analyzes the groups, then references the segments by the address in memory of that group. A program may consist on one or more groups.

If you are writing in assembly language, you may assign the group and class names in your program. In high-level languages (BASIC, COBOL, FORTRAN, Pascal), the naming is done automatically by the compiler.

2.0 FILES THAT MS-LINK USES

MS-LINK performs the following functions:

Works with one or more input files

Produces two output files

May create a temporary disk file

May be directed to search up to eight library files

For each type of file, you can give a three-part file specification. The format of MS-LINK file specifications is the same as that of a disk file:

[d:]<filename>[<.ext>]

where: d: is the drive designation. Permissible drive designations for MS-LINK are A: through O:. The colon is always required as part of the drive designation.

filename is any legal filename of one to eight characters.

.ext is a one- to three-character extension to the filename. The period is always required as part of the extension.

2.1 Input File Extensions

If no filename extensions are given in the input (object) file specifications, MS-LINK will recognize the following extensions by default:

.OBJ	Object
.LIB	Library

2.2 Output File Extensions

MS-LINK appends the following default extensions to the output (run and list) files:

.EXE	Run (may not be overridden)
.MAP	List (may be overridden)

2.3 VM.TMP (Temporary) File

MS-LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, MS-LINK will create a temporary file, name it VM.TMP, and put it on the disk in the default drive. If MS-LINK creates VM.TMP, it will display the message:

```
VM.TMP has been created.
Do not change diskette in drive, <d:>
```

Once this message has been displayed, you must not remove the disk from the default drive until the link session ends. If the disk is removed, the operation of MS-LINK will be unpredictable, and MS-LINK might display the error message:

```
Unexpected end of file on VM.TMP
```

The contents of VM.TMP are written to the file named following the Run File: prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

WARNING

> Do not use VM.TMP as a filename for any file. If you have a file named VM.TMP on the default drive and MS-LINK needs to create a VM.TMP file, MS-LINK will delete the VM.TMP already on disk and create a new VM.TMP. Thus, the contents of the previous VM.TMP file will be lost.

3.0 HOW TO START MS-LINK

MS-LINK requires two types of input: a command to start MS-LINK and responses to command prompts. In addition, seven switches control MS-LINK features. Usually, you will type all the commands to MS-LINK on the terminal keyboard. As an option, answers to the command prompts and any switches may be contained in a response file. Command characters can be used to assist you while giving commands to MS-LINK.

MS-LINK can be started in any of three ways. The first method is to type the commands in response to individual prompts. In the second method, you type all commands and switches on the line used to start MS-LINK. To start MS-LINK by the third method, you must create a response file that contains all the necessary commands, and then tell MS-LINK where that file is when you start MS-LINK.

Summary of Methods to Start MS-LINK

Method 1	LINK
Method 2	LINK <filenames>[/switches]
Method 3	LINK @<filespec>

3.1 Method 1: Prompts

To start MS-LINK with Method 1, type:

```
LINK
```

MS-LINK will be loaded into memory. MS-LINK will then display four text prompts that appear one at a time. You answer the prompts to command MS-LINK to perform specific tasks.

At the end of each line, you may type one or more switches, preceded by the switch character, a forward slash (/).

The command prompts are summarized below.

PROMPT	RESPONSES
Object Modules [.OBJ]:	List .OBJ files to be linked. They must be separated by blank spaces or plus signs (+). If a plus sign is the last character typed, the prompt will reappear. There is no default; a response is required.
Run File {.EXE]:	Give filename for executable object code. The default is first-object-filename.EXE. (You cannot change the output extension.)
List File [NUL.MAP]:	Give filename for listing. The default is NUL.MAP.
Libraries [.LIB]:	List filenames to be searched, separated by blank spaces or plus signs (+). If a plus sign is the last character typed, the prompt will reappear. The default is to search for default libraries in the object modules. (Extensions will be changed to .LIB.)

3.2 Method 2: Command Line

To start MS-LINK using Method 2, type all commands on one line. The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas. Use the following syntax:

```
LINK <object-list>,<runfile>,<listfile>,<lib-list>[/switch]
```

where: object-list is a list of object modules, separated by plus signs.

runfile is the name of the file that receives the executable output.

listfile is the name of the file that receives the listing.

lib-list is a list of library modules to be searched.

/switch refers to optional switches, which may be placed following any of the rsponse entries (just before any of the commas or after the <lib-list>, as shown).

To select the default for a field, simply type a second comma with no spaces between the two commas.

Example:

```
     LINK
     FUN+TEXT+TABLE+CARE/P/M,,FUNLIST,COBLIB.LIB
```

This command causes MS-LINK to be loaded; then the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ are loaded. MS-LINK then pauses (as a result of using the /P switch). MS-LINK links the object modules when you press any key, and produces a global symbol map (the /M switch). MS-LINK then defaults to the FUN.EXE run file; creates a list file named FUNLIST.MAP; and searches the library file COBLIB.LIB.

Method 3: Response File

To start MS-LINK with Method 3, type:

```
LINK @<filespec>
```

where: filespec is the name of a response file. A response file contains answers to the MS-LINK prompts (shown in Method 1) and may also contain any of the switches. When naming a response file, use of the filename extension is optional. Method 3 permits the command that starts MS-LINK to be entered from the keyboard or within a batch file, without requiring you to make any further responses.

To use this option, you must create a response file containing several lines of text, each of which is the response to an MS-LINK prompt. The responses must be in the same order as the MS-LINK prompts discussed in Method 1. If desired, a long response to the Object Modules: or Libraries: prompt may be typed on several lines by using a plus sign (+) to continue the same response onto the next line.

Switches and command characters can be used in the response file the same way as they are used for responses typed on the terminal keyboard.

When the MS-LINK session begins, each prompt will be displayed in order with the responses from the response file. If the response file does not contain answers for all the prompts (in the form of filenames, the semicolon command character, or carriage returns), MSLINK will display the prompt which does not have a response, then wait for you to type a legal response. When a legal response has been typed. MS-LINK continues the link session.

Example:

```
FUN TEXT TABLE CARE
/PAUSE/MAP
FUNLIST
COBLIB.LIB
```

This response file tells MS-LINK to load the four object modules named FUN, TEXT, TABLE, and CARE. MS-LINK pauses to permit you to swap disks before producing a public symbol map (see discussion under /PAUSE in the "Switches" section before using this feature). When you press any key, the output files will be named FUN.EXE and FUNLIST.MAP. MS-LINK will then search the library file COBLIB.LIB, and will use default settings for the switches.

MS-LINK recognizes three command characters:

Plus sign   Use the plus sign (+) to separate entries and to extend the current line in response to the Object Modules: and Libraries: prompts. (A blank space may be used to separate object modules.) To type a large number of responses (each may be very long), type a plus sign/<RETURN> at the end of the line to extend it. If the plus sign/<RETURN> is the last entry following these two prompts, MS-LINK will prompt you for more module names. When the Object Modules: or Libraries: prompt appears again, continue to type responses. When all the modules to be linked and libraries to be searched have been listed, be sure the response line ends with a module name and a <RETURN> and not a plus sign/<RETURN>.

Example:

```
Object Modules [.OBJ]:  FUN TEXT TABLE
CARE+<RETURN>
Object Modules [.OBJ]
FOO+FLIPFLOP+JUNQUE+<RETURN>
Object Modules [.OBJ]: CORSAIR<RETURN>
```

Semicolon

To select default responses to the remaining prompts, use a single semicolon (;) followed immediately by a carriage return at any time after the first prompt (Run File:). This feature saves time and overrides the need to press a series of <RETURN> keys.

NOTE

> Once the semicolon has been entered (by pressing the <RETURN> key), you can no longer respond to any of the prompts for that link session. Therefore, do not use the semicolon to skip some prompts. To skip prompts, use the <RETURN> key.

Example:

```
Object Modules [.OBJ]:  FUN TEXT TABLE
CARE<RETURN>
Run Module [FUN.EXT]: ;<RETURN>
```

No other prompts will appear, and MS-LINK will use the default values (including FUN.MAP for the list file).

<CONTROL-C>

Use the <CONTROL-C> key to abort the link session at any time. If you type an erroneous response, such as the wrong filename or an incorrectly spelled filename, you must press <CONTROL-C> to exit MS-LINK, then you must restart MS-LINK. If the error has been typed but you have not pressed the <RETURN> key, you may delete the erroneous characters with the backspace key, but for that line only.

5.0 MS-LINK SWITCHES

The seven MS-LINK switches control various MS-LINK functions. Switches must be typed at the end of a prompt response, regardless of which method is used to start MS-LINK. Switches may be grouped at the end of any response, or may be scattered at the end of several. If more than one switch is typed at the end of a response, each switch must be preceded by a forward slash (/).

All switches may be abbreviated. The only restriction is that an abbreviation must be sequential from the first letter through the last type; no gaps or transpositions are allowed. For example:

Legal	Illegal
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOCA	/DSALLOCT

/DSALLOCATE

/D

Using the /DSALLOCATE switch tells MS-LINK to load all data at the high end of the Data Segment. Otherwise, MS-LINK loads all data at the low end of the Data Segment. At runtime, the DS pointer is set to the lowest possible address to allow the entire DS segment to be used. Use of the /DSALLOCATE switch in combination with the default load low (that is, the /HIGH switch is not used) permits the user application to dynamically allocate any available memory below the area specifically allocated within DGroup, yet to remain addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

NOTE

Your application program may dynamically allocate up to 64K bytes (or the actual amount of memory available) less the amount allocated within DGroup.

/HIGH

/H

Use of the /HIGH switch causes MS-LINK to place the run file as high as possible in memory. Otherwise, MS-LINK places the run file as low as possible.

IMPORTANT

Do not use the /HIGH switch with Pascal or FORTRAN programs.

/LINENUMBERS

/L

The /LINENUMBERS switch tells MS-LINK to include in the list file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the list file.

NOTE

Some compilers produce object modules that do not contain line number information. In these cases, of course, MS-LINK cannot include line numbers.

/MAP

/m

/MAP directs MS-LINK to list all public (global) symbols defined in the input modules. If /MAP is not given, MS-LINK will list only errors (including undefined globals).

The symbols are listed alphabetically at the end of the list file. For each symbol, MS-LINK lists its value and its segment:offset location in the run file.

/PAUSE

/P

The /PAUSE switch causes MS-LINK to pause in the link session when the switch is encountered. Normally, MS-LINK performs the linking session from beginning to end without stopping. This switch allows the user to swap disks before MS-LINK outputs the run (.EXE) file.

When MS-LINK encounters the /PAUSE switch, it displays the message:

```
About to generate .EXE file
Change disks <hit any key>
```

MS-LINK resumes processing when you press any key.

CAUTION

Do not remove the disk which will receive the list file, or the disk used for the VM.TMP file, if one has been created.

/STACK:<number>

/S

number represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If a value from 1 to 511 is typed, MS-LINK will use 512. If the /STACK switch is not used for a link session, MS-LINK will calculate the necessary stack size automatically.

All compilers and assemblers should provide information in the object modules that allow the linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement. If not, MS-LINK will display the following error message:

```
WARNING:  NO STACK STATEMENT
```

/NO

/NO is short for NODEFAULTLIBRARYSEARCH. This switch tells MS-LINK to not search the default (product) libraries in the object modules. For example, if you are linking object modules in Pascal, specifying the /NO switch tells MS-LINK to not automatically search the library names PASCAL.LIB to resolve external references.

*

6.0 ERROR MESSAGES

All errors cause the link session to abort. After the cause has been found and corrected, MS-LINK must be rerun. The following error messages are displayed by MS-LINK:

ATTEMPT TO ACCESS DATA OUTSIDE OF SEGMENT BOUNDS, POSSIBLY BAD OBJECT MODULE

There is probably a bad object file.

BAD NUMERIC PARAMETER

Numeric value is not in digits.

CANNOT OPEN TEMPORARY FILE

MS-LINK is unable to create the file VM.TMP because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the List.Map file.

ERROR: DUP RECORD TOO COMPLEX

The DUP record in the assembly language module is too complex. Simplify the DUP record in your assembly language program.

ERROR: FIXUP OFFSET EXCEEDS FIELD WIDTH

An assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit your assembly language source and reassemble.

INPUT FILE READ ERROR

There is probably a bad object file.

INVALID OBJECT MODULE

An object module(s) is incorrectly formed or incomplete (as when assembly is stopped in the middle).

SYMBOL DEFINED MORE THAN ONCE

MS-LINK found two or more modules that define a single symbol name.

PROGRAM SIZE OR NUMBER OF SEGMENTS EXCEEDS CAPACITY OF LINKER

The total size may not exceed 384K bytes, and the number of segments may not exceed 255.

REQUESTED STACK SIZE EXCEEDS 64K
Specify a size greater than or equal to 64K bytes with the /STACK switch.

SEGMENT SIZE EXCEEDS 64K
64K bytes is the addressing system limit.

SYMBOL TABLE CAPACITY EXCEEDED
Very many and/or very long names are typed., exceeding the limit of approximately 25K bytes.

TOO MANY EXTERNAL SYMBOLS IN ONE MODULE
The limit is 256 external symbols per module.

TOO MANY GROUPS
The limit is 10 groups.

TOO MANY LIBRARIES SPECIFIED
The limit is 8 libraries.

TOO MANY PUBLIC SYMBOLS
The limit is 1024 public symbols.

TOO MANY SEGMENTS OR CLASSES
The limit is 256 (segments and classes taken together).

UNRESOLVED EXTERNALS: <list>
The external symbols listed have no defining module among the modules or library files specified.

VM READ ERROR
This is a disk error; it is not causes by MS-LINK.

WARNING: NO STACK SEGMENT
None of the object module specified contains a statement allocation stack space, although the /STACK switch was specified.

WARNING: SEGMENT OF ABSOLUTE OR UNKNOWN TYPE
There is a bad object module, or an attempt has been made to link modules that MS-LINK cannot handle (e.g., an absolute object module).

WRITE ERROR IN TMP FILE

No more disk space remains to expand the VM.TMP file.

WRITE ERROR ON RUN FILE

Usually, there is not enough disk space for the run file.

Contents

Chapter 1 **INTRODUCTION**

1.2 Overview of MS-LINK Operation 1-2
1.3 Definitions 1-4

Chapter 2 **MS-LINK TECHNICAL INFORMATION**

2.1 How MS-LINK Combines and Arranges Segments 2-1
2.2 Segment Addresses 2-4
2.3 How MS-LINK Assigns Addresses 2-4
2.4 Relocation Fixups 2-4
2.4.1 Short References 2-5
2.4.2 Near Self-Relative References 2-5
2.4.3 Near Segment-Relative References 2-6
2.4.4 Long References 2-6
2.5 Sample MS-LINK Session 2-7
2.6 Error Messages 2-9

Index

CHAPTER 1

INTRODUCTION

The Microsoft Linker Utility (MS-LINK) is a relocatable linker designed to link separately produced modules of 8086 object code. The input to MS-LINK is a subset of the Intel object module format standard.

MS-LINK prompts you for all MS-LINK commands. Your answers to these prompts are the commands for MS-LINK.

The output file from MS-LINK (a Run file) is not bound to specific memory addresses and, therefore, can be loaded and executed at any convenient address by the operating system.

MS-LINK uses a dictionary-indexed library search method, which substantially reduces link time for sessions involving library searches.

MS-LINK is able to link files totaling 1 megabyte.

NOTE

> This manual describes some of the technical information about MS-LINK. It is recommended that this manual be read in conjunction with Chapter 9, "The Linker Program (MS-LINK)," in the *MS-DOS User's Guide*.

1.1 OVERVIEW OF MS-LINK OPERATION

MS-LINK performs the following steps to combine object modules and produce a Run file:

1. Reads segments in object modules
2. Assigns addresses to segments
3. Assigns public symbol addresses
4. Reads data in segments
5. Reads all relocation references in object modules
6. Resolves references and determines relocation information
7. Outputs a Run file (executable image) and relocation information

As it combines modules, MS-LINK can search multiple library files for definitions of any external references left unresolved.

MS-LINK also produces a List file that shows external references resolved and any error messages.

MS-LINK uses available memory as much as possible. When available memory is exhausted, MS-LINK then creates a disk file (VM.TMP) to use as temporary memory.

The following figure illustrates the MS-LINK operation.

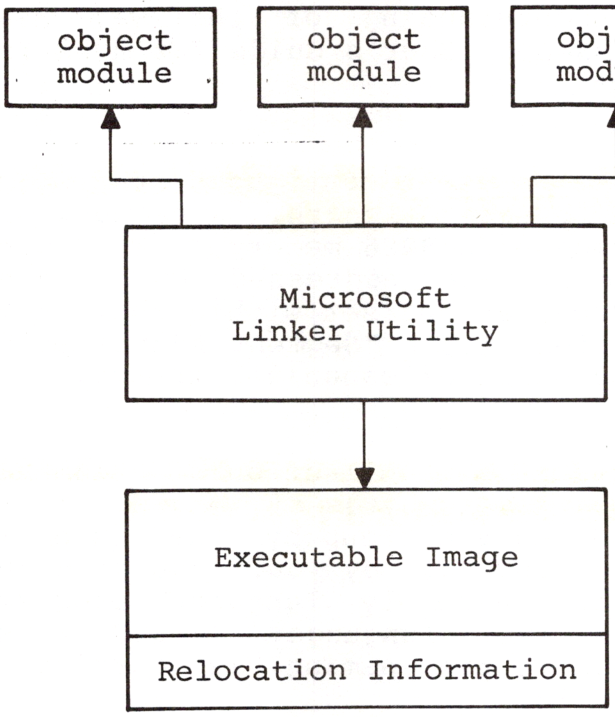

object module
object module
object module
Microsoft Linker Utility
Executable Image
Relocation Information

Figure 1. MS-LINK Operation

The *executable image* contains the concatenated object modules that make the Run file. The *relocation information* is a list of long addresses that must change when the executable image is relocated in memory. Refer to Section 1.7.4, "Long References," for an explanation of long addresses.

1.2 DEFINITIONS

The following terms describe the functioning of MS-LINK. An understanding of the concepts that define these terms will provide a basic understanding of the way MS-LINK works. Refer to the MS-DOS User's Guide for more information on these definitions.

1. Segment

   A segment is a contiguous area of memory up to 64K bytes in length. A segment may be located anywhere in 8086 memory. The contents of a segment are addressed by a canonical frame address and offset within that frame. Refer to Section 1.5, "Segment Addresses," for further discussion of canonical frames.

2. Group

   A group is a collection of segments that fit within 64K bytes of memory. The segments are named to the group by the assembler, by the compiler, or by you. You give the group name in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler.

   The group is used for addressing segments in memory. Each group is addressed by a common canonical frame. This frame is the lowest canonical frame of the segments that belong to the group. It is a usual practice in assembler and higher languages for the canonical frame address to be contained in a segment register. MS-LINK checks to see that the object modules of a group meet the 64K-byte constraint.

3. Class

   A class is a collection of segments. The naming of segments to a class controls the order and relative placement of segments in memory. You give the class name in the assembly language program. For the high-level languages (BASIC, FORTRAN, COBOL, Pascal), the naming is carried out by the compiler. The segments are named to a class at compile time or assembly time.

   The segments of a class are loaded into memory contiguously. The segments are ordered within a class in the order the Linker encounters the segments in the object files. One class precedes another in memory only if a segment for the first class precedes all segments for the second class in the input to MS-LINK. Classes may be loaded across 64K-byte boundaries. Groups may span classes.

4. Alignment

   Alignment refers to certain segment boundaries. These can be byte, word, or paragraph boundaries.

   Byte Alignment: A segment can begin on any byte boundary.

   Word Alignment: The beginning address of a segment must occur on an even address.

   Paragraph Alignment: The beginning address of a segment must occur on a segment (16-byte) boundary.

5. Combine Type

   A combine type is an attribute of a segment; it tells the Linker how to combine segments of a like name or it relays other information about the properties of a segment. Combine types are: stack, public, private, and common. The way MS-LINK arranges these combine types is discussed in the next section.

CHAPTER 2

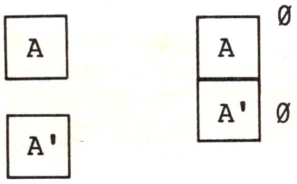
MS-LINK TECHNICAL INFORMATION

2.1 HOW MS-LINK COMBINES AND ARRANGES SEGMENTS

MS-LINK works with four combine types, which are declared in the source module for the assembler or compiler: private, public, stack, and common. The memory combine type available in Microsoft's Macro Assembler is processed the same as public combine type. MS-LINK does not automatically place memory combine type as the highest segments (as defined in the Intel standard).

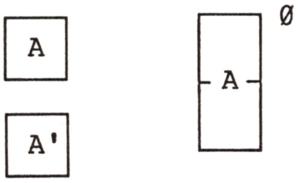
MS-LINK arranges these combine types as follows:

Private



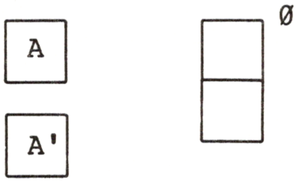
Private segments are loaded separately and remain separate. They may be physically (but not logically) contiguous even if the segments have the same name. Each private segment has its own canonical frame.

Public and Stack



Public and stack segments of the same name and class name are loaded contiguously. Offset is from the beginning of the first segment loaded through the last segment loaded. There is only one canonical frame for all public segments of the same name and class name. Stack and memory combine types are treated the same as public. However, the Stack Pointer is set to the last address of the first stack segment.

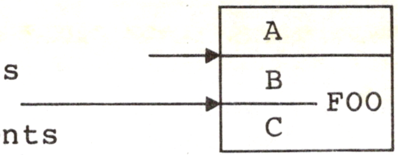
Common



Common segments of the same name and class name are loaded overlapping one another. There is only one canonical frame for all common segments of the same name. The length of the common area is the length of the longest segment.

Placing segments in a group in the assembler provides offset addressing of items from a single canonical frame for all segments in that group.

DS:DGROUP--->XXXX0H.........0 -- relative offset



Any number of other segments may intervene between segments of a group. Thus, the offset of FOO may be greater than the size of segments in the group combined, but no larger than 64K.

An operand of DGROUP:FOO in assembly language returns the offset of FOO from the beginning of the first segment (segment A here).

Segments are partitioned by declared class names. The Linker loads all the segments belonging to the first class name encountered, then loads all the segments of the next class name encountered, and so on until all classes have been loaded.

If your program contains:

```
A   SEGMENT 'FOO'
B   SEGMENT 'BAZ'
C   SEGMENT 'BAZ'
D   SEGMENT 'ZOO'
E   SEGMENT 'FOO'
```

They will be loaded as:

```
'FOO'
   A
   E
'BAZ'
   B
   C
'ZOO'
   D
```

If you are writing assembly language programs, you can control the order of classes in memory by writing a dummy module and listing it first after the MS-LINK Object Modules: prompt. The dummy module declares segments into classes in the order you want the classes loaded.

WARNING

> Do not use this method with BASIC, COBOL, FORTRAN, or Pascal programs. Allow the compiler and the Linker to perform their tasks in the normal way.

Example:

```
A       SEGMENT 'CODE'
A       ENDS
B       SEGMENT 'CONST'
B       ENDS
C       SEGMENT 'DATA'
C       ENDS
D       SEGMENT STACK   'STACK'
D       ENDS
E       SEGMENT 'MEMORY'
E       ENDS
```

Make sure you declare all classes to be used in your program in this module. If you do not, you lose absolute control over the ordering of classes.

Also, if you want memory combine type to be loaded as the last segments of your program, you can use this method. Simply add MEMORY between SEGMENT and 'MEMORY' in the E segment line above. Note, however, that these segments are loaded last only because you imposed this control on them, not because of any inherent capability in the Linker or assembler operations.

2.2 SEGMENT ADDRESSES

The 8086 must be able to address all segments in memory. Any 20-bit number can be addressed. The 8086 represents these numbers as two 16-bit numbers; for example, HEX F:12. The F represents a canonical frame address and the 12 is the offset. The canonical frame address is the largest frame address or segment address that can contain the segment. An offset is the segment's location, offset from the beginning of the canonical frame.

The Linker recognizes a segment by its canonical frame address and its offset within the frame.

To convert the segmented address F:12 to a 20-bit number, shift the frame address left 4 bits, and add the offset. Using the above example:

$$\begin{array}{rr} & F0 \\ + & 12 \\ \hline \end{array}$$

F:12 = 102 (20-bit address)

2.3 HOW MS-LINK ASSIGNS ADDRESSES

To assign addresses to segments, MS-LINK:

1. Orders each segment by segment and class name.

2. On the basis of the alignment and size of each segment (assuming they are contiguous), the Linker assigns a frame address and an offset to each segment. This information is used for resolving relocatable references. The addresses start at 0:0.

2.4 RELOCATION FIXUPS

MS-LINK performs relocation fixups (i.e., resolves) on four types of references in object modules:

Short

Near Self-Relative

Near Segment-Relative

Long

These references and the Linker's fixups are described in the next sections.

2.4.1 Short References (+ -) 128 bytes

Short references are all self-relative. The implication is that the frame address of the target and source frames are the same. MS-LINK will generate the fixup error message

```
Fixup offset exceeds field width
```

under the following conditions:

1. The target and source frame addresses are different.

2. The target is more than 128 bytes before or after the source frame address.

The resulting value of the short reference must fit into one signed byte.

2.4.2 Near Self-Relative References (+ -) 32K bytes

When near self-relative references are used, the frame address of the target and source frames are the same. MS-LINK will generate the fixup error message under the following conditions:

1. The target and source frame addresses are different.

2. The target is more than 32K before or after the source frame address.

The resulting value of the near self-relative reference must fit into one signed word (16 bits).

2.4.3 Near Segment-Relative References

0 - 64K

Given the target's canonical frame, another frame is specified (via an ASSUME directive or the : operator in assembly language; or via a high-level language convention). The target must be addressable through the canonical frame specified. MS-LINK will generate the fixup error message under the following conditions:

1. The offset of the target within the specified frame is greater than 64K or less than zero.
2. The beginning of the canonical frame of the target is not addressable by the specified frame.

The resulting value of a near segment-relative reference must be an unsigned 16-bit quantity.

2.4.4 Long References

Long references have a target and another frame (specified by an ASSUME or by a high-level language). The target must be addressable through the canonical frame specified. MS-LINK will generate the fixup error message under the following conditions:

1. The offset of the target within the specified frame is greater than 64K or less than zero.
2. The beginning of the canonical frame of the target is not addressable by the specified frame.

The resulting value of a long reference must be a frame address and an offset.

*

2.5 SAMPLE MS-LINK SESSION

The following example illustrates the type of information that is displayed during an MS-LINK session.

In response to the MS-DOS prompt (>), the system responds with the following messages and prompts. Answers to the prompts are underlined. Note that pathnames are supported under MS-DOS 2.0. Therefore, your answers to MS-LINK prompts can be full pathnames instead of filenames.

```
Microsoft Object Linker V.2.00
(C) Copyright 1982 by Microsoft Inc.

Object Modules ].OBJ[: IO SYSINIT
Run File ]IO.EXE[:
List File ]NUL.MAP[: IO /MAP
Libraries ].LIB[: ;
```

Notes:

1. By specifying /MAP, you can get both a sorted alphabetic listing and a sorted address listing of public symbols. ←

2. By responding PRN to the List File: prompt, you can redirect your output to the printer.

3. By specifying the /LINE switch, MS-LINK gives you a listing of all line numbers for all modules. (Note that the /LINE switch can generate a large volume of output.)

4. By pressing <RETURN> in response to the Libraries: prompt, an automatic library search is performed.

Once MS-LINK locates all libraries, the linker map displays a list of segments in the order of their appearance within the load module. The list might look like this:

```
Start     Stop      Length     Name
00000H    009ECH    09EDH      CODE
009F0H    01166H    0777H      SYSINITSEG
```

The information in the Start and Stop columns shows the 20-bit hex address of each segment relative to location zero. Location zero is the beginning of the load module.

Because the /MAP switch was used, MS-LINK displays the public symbols by name and value. For example:

```
ADDRESS             PUBLICS BY NAME
009F:0012           BUFFERS
009F:0005           CURRENT_DOS_LOCATION
009F:0011           DEFAULT_DRIVE
009F:000B           DEVICE_LIST
009F:0013           FILES
009F:0009           FINAL_DOS_LOCATION
009F:000F           MEMORY_SIZE
009F:0000           SYSINIT

ADDRESS             PUBLICS BY VALUE
009F:0000           SYSINIT
009F:0005           CURRENT_DOS_LOCATION
009F:0009           FINAL_DOS_LOCATION
009F:000B           DEVICE_LIST
009F:000F           MEMORY_SIZE
009F:0011           DEFAULT_DRIVE
009F:0012           BUFFERS
009F:0013           FILES
```

The addresses of the public symbols are in the frame:offset format, showing the location relative to zero as the beginning of the load module. In some cases, an entry may look like this:

```
780:A2
```

This entry appears to be the address of a load module that is almost one megabyte in size. Actually, the area being referenced is relative to a segment base that is pointing to a segment below the relative zero beginning of the load module. This condition produces a pointer that has effectively gone negative.

When MS-LINK has completed processing, the following message is displayed:

```
Program entry point at 0009F:0000
```

2.6 ERROR MESSAGES

All messages, except for the warning messages, cause the MS-LINK session to end. After you locate and correct a problem, you must rerun MS-LINK.

Messages appear in the List file and are displayed on the screen. If you direct the List file to CON, the error messages will not be displayed on the screen.

MS-LINK error messages are described in Chapter 9 of the MS-DOS User's Guide.

INDEX

Alignment 1-5

Canonical frame address . . . 2-4
Class 1-4
Class names 2-2
Combine type 1-5

Error messages 2-9
Executable image . . . . 1-2 to 1-3

Fixup error 2-5

Group 1-4

How MS-LINK combines and arranges segments 2-1

Library files 1-2
List file 1-2
Long addresses 1-3
Long references 2-6

Near segment-relative references 2-6
Near self-relative references 2-5

Offset 2-4
Offset addressing 2-2
Overviews
  MS-LINK operation 1-2

Pathnames 2-7
Public symbols 2-8

Relocation fixups 2-5
  long 2-5
  near segment-relative 2-5
  near self-relative 2-5
  short 2-5
Run file . . . . 1-1 to 1-3

Sample MS-LINK session 2-7
Segment 1-4
Segment addresses 2-4
Short references 2-5

VM.TMP 1-2

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® LIB

Library Manager

for 8086 and 8088 Microprocessors

Preliminary

LIB
Manager

SEATTLE
COMPUTER
1114 Industry Dr. Seattle WA 98188 206/575-1830

Microsoft® LIB

Library Manager

for 8086 and 8088 Microprocessors

Preliminary

System Requirements

The Microsoft LIB Library Manager requires:

38K bytes of memory minimum:
    28K bytes for code
    10K bytes for run space

Disk drive(s):
    One disk drive if and only if output is sent to the same physical disk from which the input was taken. The Microsoft LIB Library Manager does not allow time to swap disks during operation on a one-drive configuration. Therefore, two disk drives is a more practical configuration.

Contents

Chapter 1 INTRODUCTION

1.1 Features of MS-LIB 1-1
1.2 Overview of MS-LIB Operation 1-2

Chapter 2 RUNNING MS-LIB

2.1 How to Start MS-LIB 2-1
2.1.1 Method 1: Prompts 2-2
2.1.2 Method 2: Command Line 2-3
2.1.3 Method 3: Response File 2-5
2.2 Command Prompts 2-7
2.3 Command Characters 2-9
+ - append 2-9
- - delete 2-9
* - extract 2-10
; - default remaining prompts 2-10
& - continuation 2-11
CONTROL-C - program abort 2-12

Chapter 3 ERROR MESSAGES

Index

CHAPTER 1

INTRODUCTION

1.1 FEATURES OF MS-LIB

Microsoft LIB is a library manager. With MS-LIB, you can:

Create and modify library files that are used with Microsoft's MS-LINK linker utility

Add object files to a library

Delete modules from a library

Extract modules from a library and place the extracted modules into separate object files

MS-LIB can create either general or special libraries, for a variety of programs or for specific programs. With MS-LIB you can create a library, or you can create a library for one program only. The result is fast linking and more efficient execution for a language compiler or for one program.

You can modify individual modules within a library by extracting the modules, making changes, then adding the modules to the library again. You can also replace an existing module with a different module or with a new version of an existing module.

The command scanner in MS-LIB is also used in Microsoft MS-LINK, MS-Pascal, MS-FORTRAN, and other 16-bit Microsoft products. If you have used any of these products, using MS-LIB should be familiar to you. Command syntax is straightforward, and MS-LIB prompts you for commands that you have not supplied.

1.2 OVERVIEW OF MS-LIB OPERATION

MS-LIB performs five library manager functions:

Deletes modules

Extracts a module and places it in a separate object file

Appends an object file as a module of a library

Replaces a module in the library file with a new module

Creates a library file

During each library session, MS-LIB deletes or extracts modules, then appends new ones to the library file. MS-LIB reads each module into memory, checks it for consistency, and writes it back to the file. If you delete a module, MS-LIB reads that module into memory but does not write it back to the file. When MS-LIB writes back the next module to be retained, it places that module at the end of the last module written. This procedure effectively "closes up" the disk space to keep the library file from growing too large.

When MS-LIB has read the library file, it appends any new modules to the end of the file. Finally, MS-LIB creates the index, which MS-LINK uses to find modules and symbols in the library file. MS-LIB will output a cross-reference listing of the PUBLIC symbols in the library, if you request such a listing.

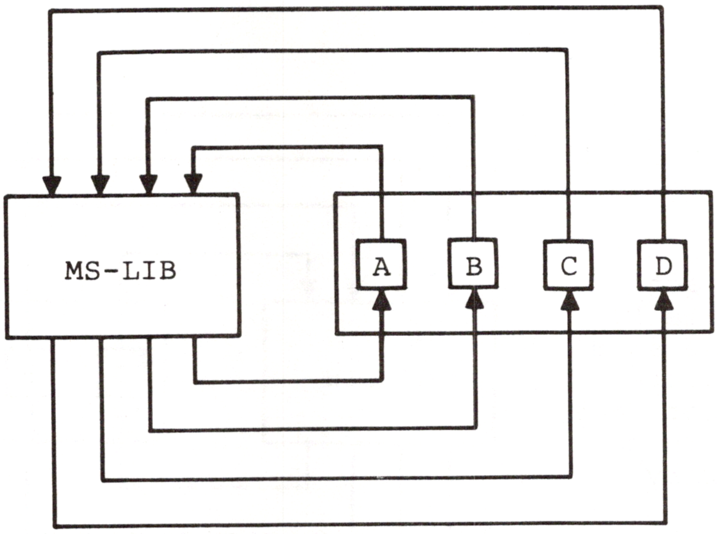
Example:

```
LIBx PASCAL+HEAP-HEAP;
```

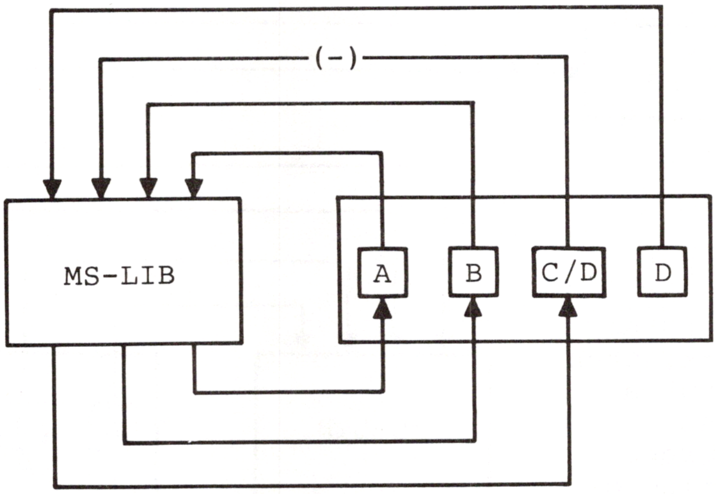
This command first deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. Note that the replace function is simply the delete-append functions in succession. Also note that you can specify delete, append, or extract functions in any order. This order of execution prevents confusion in MS-LIB when a new version of a module replaces a version in the library file.

The following figure illustrates the MS-LIB operation.

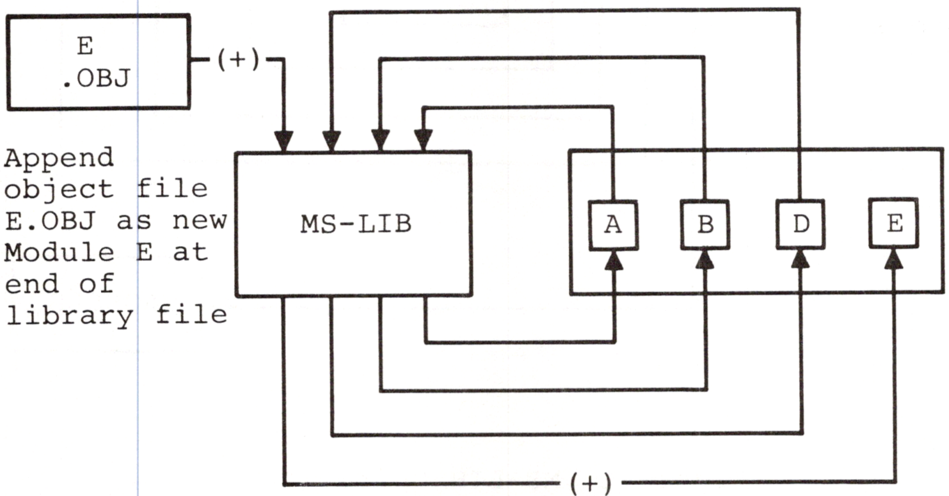
Consistency
check only

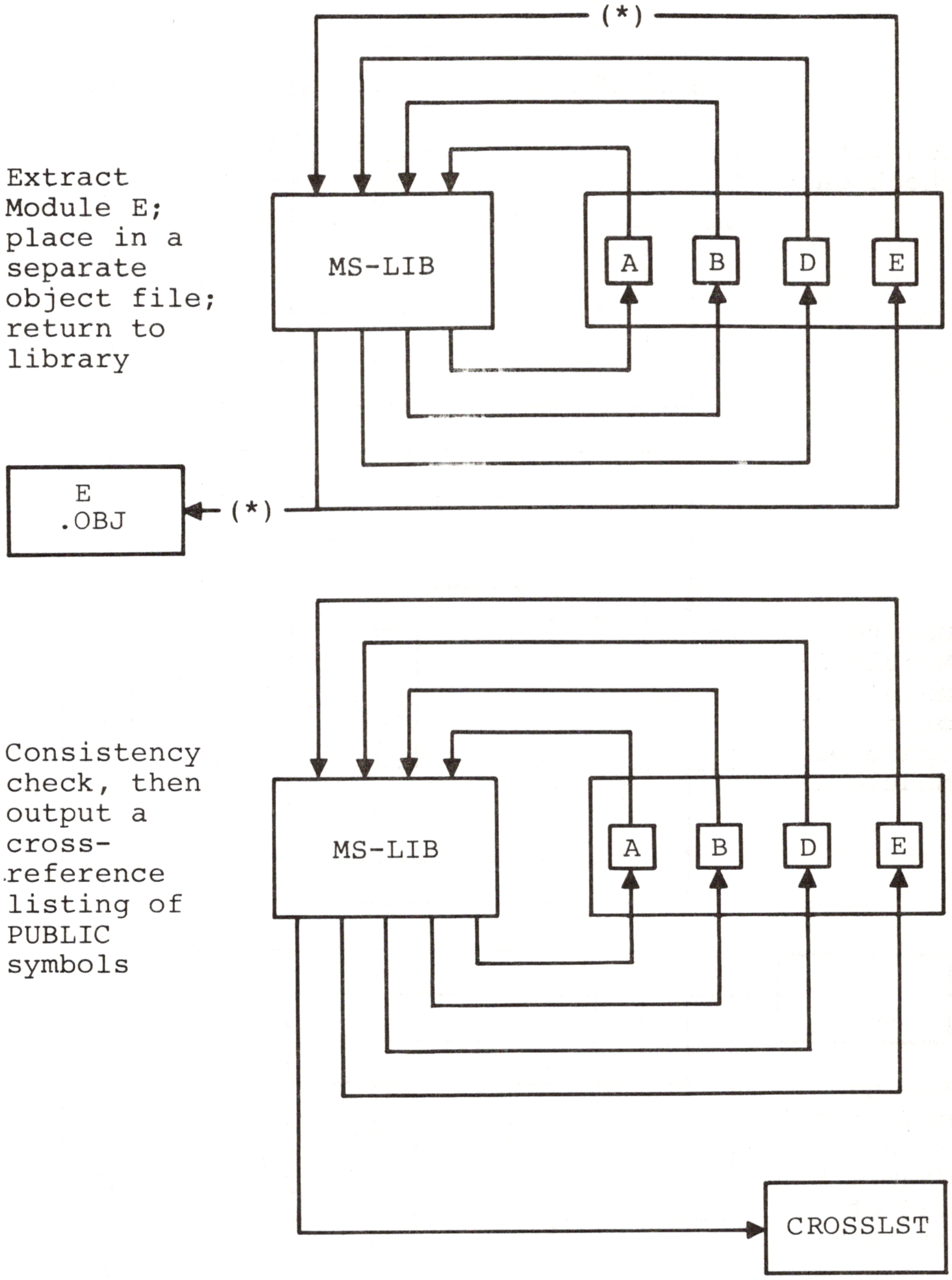


Delete
Module C;
Module D
written to
space of
Module C


(-)
MS-LIB
A
B
C/D
D

Append
object file
E.OBJ as new
Module E at
end of
library file


E
.OBJ
(+)
MS-LIB
A
B
D
E
(+)



Extract Module E; place in a separate object file; return to library

Consistency check, then output a cross-reference listing of PUBLIC symbols

(*)
MS-LIB
A
B
D
E
E .OBJ
(*)
MS-LIB
A
B
D
E
CROSSLST

Figure 1. MS-LIB Operation

CHAPTER 2

RUNNING MS-LIB

Running MS-LIB requires two types of commands: a command to start MS-LIB and answers to command prompts. Usually you will type all the commands to MS-LIB on a command line or in response to MS-LIB prompts. As an option, answers to the command prompts may be contained in a response file. Command characters can be used to assist you while giving commands to MS-LIB.

2.1 HOW TO START MS-LIB

There are three ways to start MS-LIB. With the first method, you type the commands as answers to individual prompts. By the second method, you type all commands on the line used to start MS-LIB. As a third option, you can create a response file that contains all the necessary commands.

Summary of Methods to Start MS-LIB

Method 1	LIB
Method 2	LIB <library><operations>,<listing>
Method 3	LIB #<filespec>

2.1.1 Method 1: Prompts

To start MS-LIB with method 1, type:

```
LIB
```

MS-LIB will be loaded into memory. MS-LIB will then display three text prompts that appear one at a time. You answer the prompts, commanding MS-LIB to perform specific tasks.

The command prompts are summarized here and described more fully in the Section 2.2, "Command Prompts."

Summary of Command Prompts

PROMPT	RESPONSES
Library File:	List filename of library to be manipulated. (The default is the filename extension .LIB.)
Operation:	List command character(s) followed by module name(s) or object filename(s). (The default is no changes. The default object filename extension is .OBJ.)
List file:	List filename for a cross-reference listing file. (The default is NUL; i.e., no file.)

NOTE

The distinction between an object file and a module (or object module) is that the file possesses a drive designation (even if it is the default drive) and a filename extension. Object modules possess neither of these.

2.1.2 Method 2: Command Line

Type:

```
LIB <library><operations>,<listing>
```

The entries following LIB are responses to the command prompts. The <library> and <operations> fields and all operations entries must be separated by one of the command characters plus, minus, or asterisk (+, -, or *). If a cross-reference listing is wanted, the name of the file must be separated from the last operations entry by a comma.

where: <library> is the name of a library file. MS-LIB assumes that the filename extension is .OBJ, which you may override by specifying a different extension. If the filename given for the <library> field does not exist, MS-LIB will prompt you:

```
Library file does not exist. Create?
```

Type Yes to create a new library file. Type No to abort the library session.

<operations> is a command to delete a module, append an object file as a module, or extract a module as an object file from the library file. Use the three command characters plus, minus, and asterisk to direct MS-LIB to append, delete, or extract modules and object files.

<listing> is the name of the file you want to receive the cross-reference listing of PUBLIC symbols in the modules in the library. The list is compiled after all module manipulation has taken place.

If you type a library filename followed immediately by a semicolon, MS-LIB will read through the library file and perform a consistency check. No changes will be made to the modules in the library file.

If you type a library filename followed immediately by a comma and a listing filename, MS-LIB will perform its consistency check of the library file, then produce the cross-reference listing file.

Examples:

```
LIB PASCAL-HEAP+HEAP;
```

This example causes MS-LIB to delete the module HEAP from the library file PASCAL.LIB, then append the object file HEAP.OBJ as the last module of PASCAL.LIB (the module will be named HEAP). The MS-LIB semicolon command character indicates that MS-LIB should use the default responses for the remaining prompts. Refer to Section 2.3, "Command Characters," for more information.

```
LIB PASCAL
```

This example causes MS-LIB to perform a consistency check of the library file PASCAL.LIB. No other action is performed.

```
LIB PASCAL,PASCROSS.PUB
```

This example causes MS-LIB to perform a consistency check of the library file PASCAL.LIB, then output a cross-reference listing file named PASCROSS.PUB.

If you have many operations to perform during a library session, use the ampersand (&) command character to extend the line so that you can type additional object filenames and module names. Be sure to always include one of the command characters for operations (+, -, *) before the name of each module or object filename.

2.1.3 Method 3: Response File

Type:

```
LIB #<filespec>
```

where: <filespec> is the name of a response file. A response file contains answers to the MS-LIB prompts. Method 3 permits you to conduct the MS-LIB session without user responses to the MS-LIB prompts.

IMPORTANT

Before using method 3 to start MS-LIB, you must first create a response file.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts appear.

Use command characters in the response file the same way you would for responses typed on the keyboard.

When the library session begins, each prompt will be displayed with the responses from the response file. If the response file does not contain answers for all the prompts, MS-LIB will use the default responses. (No changes will be made to the modules currently in the library file, and no cross-reference listing file will be created.)

If you type a library filename followed immediately by a semicolon, MS-LIB will read through the library file and perform a consistency check. No changes will be made to the modules in the library file.

If you type a library filename, a carriage return, a comma, and then a list filename, MS-LIB will perform its consistency check of the library file, then produce the cross-reference listing file.

Example:

```
PASCAL
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes MS-LIB to delete the module HEAP from the PASCAL.LIB library file; extract the module FOIBLES and place it in an object file named FOIBLES.OBJ; then append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Then, MS-LIB will create a cross-reference file named CROSSLST.

2.2 COMMAND PROMPTS

You command MS-LIB by typing responses to three text prompts. After you have typed your response to the current prompt, the next appears. When the last prompt has been answered, MS-LIB performs its library management functions without further command. You will see the operating system prompt when MS-LIB has finished the library session successfully. If the library session is unsuccessful, MS-LIB will display the appropriate error message.

MS-LIB prompts you for the name of the library file, the operation(s) you want to perform, and the name you want to give to a cross-reference listing file (if any).

Command Prompts

Library File:

Type the name of the library file that you want to manipulate. MS-LIB assumes that the filename extension is .LIB. You can override this assumption by giving a filename extension when you type the library filename. Because MS-LIB can manage only one library file at a time, only one filename is allowed in response to this prompt. Additional responses, except the semicolon command character, are ignored.

If you type a library filename and follow it immediately with a semicolon command character, MS-LIB will perform a consistency check only, then return to the operating system. Any errors in the file will be displayed.

If the filename you type does not exist, MS-LIB will display the prompt:

```
Library file does not exist.  Create?
```

You must type either Yes or No.

Operation:

Type one of the three command characters for manipulating modules (+, -, *); followed immediately (no space) by the module name or the object filename. The plus sign appends an object file as the last module in the library file (see further discussion under the description of plus sign in the next section). The minus sign deletes a module from the library file. The asterisk extracts a module from the library and places it in a separate object file, with the filename taken from the module name and a filename extension .OBJ.

When you have a large number of modules to manipulate (more than can be typed on one line), type an ampersand (&) as the last character on the line. MS-LIB will repeat the Operation: prompt, which permits you to type additional module names and object filenames.

MS-LIB allows you to perform operations on modules and object files in any order you want.

More information about modules is given in the description of each command character.

List file:

If you want a PUBLIC symbols cross-reference list for the modules in the library file, type the name of a file in which you want MS-LIB to place the cross-reference listing. If you do not type a filename, no cross-reference listing is generated.

The response to the List file: prompt is a file specification. You can specify a drive (or device) designation and a filename extension with the filename. The list file is not given a default filename extension. If you want the file to have a filename extension, you must specify it when typing the filename.

The cross-reference listing file contains two lists. The first list is an alphabetical listing of all PUBLIC symbols. Each symbol name is followed by the name of its module. The second list is an alphabetical list of the modules in the library. Under each module name is an alphabetical listing of the PUBLIC symbols in that module.

2.3 COMMAND CHARACTERS

MS-LIB provides six command characters. Three of the command characters are required in response to the Operation: prompt. The other three command characters provide you with helpful commands to MS-LIB.

Plus sign	Use the plus sign (+), followed by an object filename, to append the object file as the last module in the library named in response to the Library File: prompt. When MS-LIB sees the plus sign, it assumes that the filename extension is .OBJ. You may override this assumption by specifying a different filename extension.
	MS-LIB strips the drive designation and the extension from the object file specification, leaving only the filename. For example, if the object file to be appended as a module to a library is
	`B:CURSOR.OBJ`
	a response to the Operation: prompt of
	`+B:CURSOR.OBJ`
	will cause MS-LIB to strip off the B: and the .OBJ, leaving only CURSOR. This becomes a module named CURSOR in the library.

Minus sign	Use the minus sign, followed by a module name, to delete a module from the library file. MS-LIB then "closes up" the disk space left empty by the deletion. This cleanup action keeps the library file from growing larger than necessary. Remember that new modules, even replacement modules, are added to the end of the file, not put into space vacated by deleting modules.

Asterisk

Use the asterisk, followed by a module name, to extract the module from the library file and place it into a separate object file. The module will still exist in the library. (The extraction process copies the module to a separate object file.) The module name is used as the filename. MS-LIB adds the default drive designation and the filename extension .OBJ. For example, if the module to be extracted is

```
CURSOR
```

and the current default disk drive is A:, a reponse to the Operation: prompt of

```
*CURSOR
```

causes MS-LIB to extract the module named CURSOR from the library file and make it an object file with the file specification of:

```
A:CURSOR.OBJ
```

The drive designation and filename extension cannot be overridden. You can, however, rename the file, giving a new filename extension; and/or copy the file to a new disk drive, giving a new filename and/or filename extension.

Semicolon

Use a single semicolon (;), followed immediately by a carriage return at any time after responding to the first prompt (i.e., from Library File: on), to select default responses to the remaining prompts. This feature saves time and overrides the need to answer additional prompts.

NOTE

> Once the semicolon has been typed, you can no longer respond to any of the prompts for that library session. Therefore, do not use the semicolon to skip over prompts. To skip prompts, use carriage return.

Example:

```
Library file:  FUN
Operation:  +CURSOR;
```

The remaining prompt will not appear, and MS-LIB will use the default value (no cross-reference file).

Ampersand

Use the ampersand to extend the current line. This command character is only used in response to the Operation: prompt. The number of modules you can append is limited only by disk space. The number of modules you can replace or extract is also limited only by disk space. The number of modules you can delete is limited by the number of modules in the library file.

The line length for a response to any prompt is limited to the line length of your system. For a large number of responses to the Operation: prompt, place an ampersand at the end of a line. MS-LIB will display the Operation: prompt again, and then you can type more responses. For example:

```
Library File:  FUN
Operation:  +CURSOR-HEAP+HEAP*FOIBLES&
Operation:  *INIT+ASSUME+RIDE;
```

MS-LIB will delete the module HEAP; extract the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ); then append the object files CURSOR, HEAP, ASSUME, and RIDE. Note that MS-LIB allows you to type your Operation: reponses in any order. You may use the ampersand character as many times as needed.

CONTROL-C	Use <CONTROL-C> to abort the library session at any time. If you type an incorrect response, such as the wrong filename or module name, or an incorrectly spelled filename or module name, you must press <CONTROL-C> to exit MS-LIB; then you must restart MS-LIB. If the error has been typed and you have not pressed the <RETURN> key, you may delete the erroneous characters for that line only.

Summary of Command Characters

Character	Action
+	Appends an object file as the last module
-	Deletes a module from the library
*	Extracts a module and places in an object file
;	Use default responses to remaining prompts
&	Extends current physical line; repeats command prompt
CONTROL-C	Aborts library session

CHAPTER 3

ERROR MESSAGES

The following are MS-LIB error messages:

<symbol> is a multiply defined PUBLIC. Proceed?
Cause: Two modules define the same public symbol. You are asked to confirm the removal of the definition of the old symbol.

Cure: Remove the PUBLIC declaration from one of the object modules and recompile or reassemble. If you respond No, the library will be left in an indeterminate state.

Allocate error on VM.TMP
Cause: Out of disk space

Cannot create extract file
Cause: No room in directory for extract file

Cannot create list file
Cause: No room in directory for library file

Cannot nest response file
Cause: #filespec in response (or indirect) file

MS-LIB cannot open VM.TMP
Cause: There is no room for VM.TMP in disk directory

Cannot write library file
Cause: Out of disk space

Close error on extract file
Cause: Out of disk space

Error: An internal error has occurred
Contact Microsoft Corporation

Fatal Error: Cannot open input file
Cause: You mistyped an object filename

Fatal Error: Module is not in the library
Cause: You tried to delete a module that is not in the library

Input file read error
Cause: Bad object module or faulty disk

Invalid object module/library
Cause: Bad object module and/or library

Library Disk is full
Cause: No more room on disk

Listing file write error
Cause: Out of disk space

No library file specified
Cause: No response to Library File: prompt

Read error on VM.TMP
Cause: Disk not ready for read

Symbol table capacity exceeded
Cause: Too many public symbols (about 30K chars in symbols)

Too many object modules
Cause: More than 500 object modules

Too many public symbols
Cause: 1024 public symbols maximum

Write error on library/extract file
Cause: Out of disk space

Write error on VM.TMP
Cause: Out of disk space

INDEX

& (command character) 2-11, 2-13
* (command character) 2-10, 2-13
+ (command character) 2-9, 2-13
- (command character) 2-9, 2-13
; (command character) 2-10, 2-13

Command Characters 2-9
  & 2-11, 2-13
  * 2-10, 2-13
  + ... 2-9, 2-13
  - 2-9, 2-13
  ; 2-13
  CONTROL-C 2-12
  Control-C 2-13
  Summary of 2-13
Command Prompts 2-7
  Library file 2-2, 2-7
  List file 2-2, 2-8
  Operation 2-2, 2-8
  Summary of 2-2
Consistency check 2-3, 2-5
CONTROL-C (command character) 2-12
Control-C (command character) 2-13
Creating a new library 2-3, 2-7

Error messages 3-1

Library file (command prompt) 2-2, 2-7
List file (command prompt) . . 2-2, 2-8

Method 1 2-2
Method 2 2-3
Method 3 2-5

Operation (command prompt) . . 2-2, 2-8
Overviews
  MS-LIB operation 1-2

Response File 2-5
Running MS-LIB 2-1

Starting
  Method 1 2-2
  Method 2 2-3
  Method 3 2-5
  Summary of Methods 2-1
Starting MS-LIB 2-1
Summary of methods to start . 2-1